MCCI
3520 Krums Corners Road
Ithaca, New York 14850 USA
Phone +1-607-277-1029
Fax +1-607-277-6844
www.mcci.com

USBRC User's Guide

Engineering Report 950061
Rev. J
Date: 08/01/2006

Copyright © 2006
All rights reserved

PROPRIETARY NOTICE AND DISCLAIMER

Unless noted otherwise, this document and the information herein disclosed are proprietary to Moore Computer Consultants, Incorporated, 3520 Krums Corners Road, Ithaca, New York 14850 ("MCCI"). Any person or entity to whom this document is furnished or having possession thereof, by acceptance, assumes custody thereof and agrees that the document is given in confidence and will not be copied or reproduced in whole or in part, nor used or revealed to any person in any manner except to meet the purposes for which it was delivered. Additional rights and obligations regarding this document and its contents may be defined by a separate written agreement with MCCI, and if so, such separate written agreement shall be controlling.

The information in this document is subject to change without notice, and should not be construed as a commitment by MCCI. Although MCCI will make every effort to inform users of substantive errors, MCCI disclaims all liability for any loss or damage resulting from the use of this manual or any software described herein, including without limitation contingent, special, or incidental liability.

MCCI, TrueCard, TrueTask, MCCI Catena, and MCCI USB DataPump are registered trademarks of Moore Computer Consultants, Inc.

MCCI Instant RS-232, MCCI Wombat and InstallRight Pro are trademarks of Moore Computer Consultants, Inc.

All other trademarks and registered trademarks are owned by the respective holders of the trademarks or registered trademarks.

Copyright © 2006 by Moore Computer Consultants, Incorporated

Document Release History

Rev A	7/24/ 1997	Initial release for USBRC V1.24a.
Rev B	7/28/1997	Correct typos, for USBRC V1.25a
Rev C	9/15/1997	Update for USBRC V1.26a; example output changed due to correction of a bug in string descriptor handling (MCCI SPR #0046).
Rev D	8/4/1998	Update for USBRC V2.00h
Rev E	9/3/1998	Editorial corrections.
Rev F	9/21/1998	Update to add string index header generation, info for V2.10a
Rev G	3/19/2001	add descriptor info and language codes; changes between V2.10a and V2.11a. CDC Ethernet adapter example.
Rev H	2/10/2003	Update for V2.22. Add `-dpapi` and `-chipinfo` switches. "DataPump" is now the preferred keyword for the URC file section that provides MCCI USB DataPump-specific information.
Rev I	2/27/2006	Update for V2.23b. Add On-The-Go statement under USB-resource-file and an example.
Rev. J	8/1/2006	Update for V2.23n. Add Expression evaluation for high speed support

TABLE OF CONTENTS

1 Introduction 1

1.1 Q&A 1

2 The Layout of USBRC files 4

2.1 The Basics 4
2.1.1 Identifiers 4
2.1.2 Keywords 4
2.1.2.1 Special Keywords in the "DataPump" Section 5
2.1.2.2 Special Keywords in the "Application" Section 5
2.1.2.3 Special Keywords in the "On-The-Go" Section 5
2.1.3 Comments 5
2.1.4 Numbers 6
2.1.5 Strings 6
2.1.6 Expression 8
2.1.6.1 Operators 9

2.2 The Input Language 10

2.3 The “USB-resource-file” Section 10
2.3.1 The “USB-device” Statement 10
2.3.2 The “Configuration” Statement 11
2.3.3 The “Interface” Statement 12
2.3.4 The “Endpoints” Statement 12
2.3.5 The “Strings” Statement 13
2.3.6 The “Private-Descriptor” Statement 1
2.3.7 The “Data” Statement 2
2.3.8 The “On-The-Go” Statement 2

2.4 The “DataPump” Section 2

2.5 The “Application” Section 4

2.6 String Expressions in USBRC 6

3 Using the output 7

4 Command Line Reference 7

5 Example resource description files 12

5.1 A Simple Device 12

5.2 A Device with Multiple Interfaces 16

5.3 A Multiple Interface Device, With Strings For Several Languages 21

5.4 Multiple Configurations and Private Data 28

5.5 Sample DataPump section 32

5.6 Sample Application Section 33

5.7 Sample CDC USB Ethernet Adapter 34

5.8 A Simple Device with OTG 37

5.9 A Simple Device with High speed / Expressions 39

LIST OF TABLES

Table 1. String Values in Application Section 5

LIST OF FIGURES

Error! No table of figures entries found.

1 Introduction

This document describes and specifies version V2.23n of the USB Resource Compiler (USBRC), developed by MCCI. USBRC simplifies the development of USB peripherals, by generating data for USB descriptors. USBRC reads a text file containing a high-level description of a particular USB device, and produces another file containing several tables of data, suitable for feeding to an ANSI C compiler. At runtime, the device firmware passes the data to the host in response to GET DESCRIPTOR commands, transmitting the data directly from ROM.

When used with the MCCI USB DataPump®, USBRC also simplifies peripheral design by automatically generating the header files and initialization code needed by the DataPump code.

1.1 Q&A

Why is a resource compiler needed?

USB devices must return predefined data structures in response to GET DESCRIPTOR commands from the host. These descriptors are, simply put, unwieldy to develop and maintain.

- Some operating systems are extremely sensitive to errors in the USB resource data files, and will crash while attempting to recognize a device that returns erroneous data.

- The string and configuration descriptors are variable-length, as is the endpoint descriptor in pending revisions of the specification. Standard C has no convenient way to place the needed data in ROM. USBRC automatically determines the length for each descriptor, and places the data in the appropriate places in the descriptor table.

- When working in portable C, creating the Unicode string tables is awkward at best.

- The USB specification provides for devices that store the string descriptors in many languages. If one wishes to create such devices, it is difficult to find portable tools that will generate the required Unicode in a format that is useful on any microprocessor.

Because of these issues, MCCI developed a special program, the USB Resource Compiler (USBRC). Instead of hand coding the tables with Chapter 9 of the USB specifications in hand, the compiler allows a programmer to describe USB devices in plain text. The compiler then translates the descriptive files into C data initialization statements for data structures that are intended to be placed in ROM.

Embedded macros often have very stringent storage requirements. How can one program meet all these needs?

The produced code is intended to be used with a C macro package. A sample package is provided on the distribution disk which you should edit with a text editor to meet your requirements. USBRC makes certain assumptions about how best to represent the data structures. Usually these assumptions are correct. If they are not, you can edit the file produced by USBRC to meet your needs.

But I don't want to use C. I need to use assembler.

There are at least two ways to do this.

- If you have a C compiler you can use it to compile the data structures (only), and link the resulting file with your assembly code.

- You can use SED or a manual text editor to convert the output into the format your assembler requires.

What about changes in the standard? Microsoft, for example, has requested a new power-management descriptor embedded in the data returned by Request Configuration Descriptor. If I use USBRC, can I represent the descriptor?

Yes. At various places, you can insert "**`private-descriptor`**" clauses, which allow you to enter custom descriptors at the appropriate place in the resulting data stream. Of course, you must hand-format the body of the descriptor, but USBRC will take care of setting the length for you.

How do I specify a USB devices using USBRC?

The resource compiler reads a simple language input file that is divided into sections that track the USB specification.

- The "**`device`**" section includes the information that goes into the Device Descriptor. A device only has one Device Descriptor, so there is only one "**`device`**" section.

- For each configuration of the device, you must write a "**`configuration`**" section. Each "**`configuration`**" section contains the information that goes into that configuration's Configuration Descriptor, along with the associated Interface and Endpoint descriptors. Each "**`configuration`**" has one or more "**`interface`**" clauses, and each "**`interface`**" clause includes "**`endpoint`**" descriptions of each of the endpoints available when the host selects the containing configuration and then selects (if necessary) the appropriate alternate interface setting.

- If you only want to include strings for a single language, you can put the strings in the appropriate places in the device, configuration, and interface descriptors; USBRC will automatically create string descriptors and keep track of string IDs.

- If you want to include strings for multiple languages, you will need to include a "**`strings`**" section. The "**`strings`**" section is divided into sections that are associated with specific languages. You can specify the language by name or use a numeric language code as specified by the USB specification. SEE EXAMPLES AT SECTION 5, PAGE 12.

How do I enter UNICODE strings?

If your strings consist of characters that are part of the 7-bit ASCII set, you can simply enter the strings in quotes. If you are entering characters outside the ASCII range, you have several choices:

- prepare the text file in UNICODE rather than in ASCII;
- prepare the file in UTF-8, UTF-16, or ISO-10646; or
- look up the UNICODE character values, and enter the hexadecimal values using a special notation.

What is "UTF-8"? How does it differ from Unicode? Why does USBRC use it?

Unicode is a 16-bit means of representing a wide range of characters. Unicode can represent about 60,000 characters using a 16-bit codeword, and another million characters using two Unicode codewords. All the character values are selected from the 31-bit space defined by the ISO in their standard ISO-10646. For most Unicode characters (those in which one codeword represents one character), the Unicode encoding is the same as the ISO-10646 code. UTF-8 is an ISO-10646-standardized method of representing ISO-10646 character values using an 8-bit codeword. Each character value maps into one or more codewords. UTF-8 is particularly convenient for use in most current computing environments, because the first 128 codes are identical to ASCII. Only the codes between 0x80 and 0xFF differ from their normal interpretations; they are used (only) to make 2- to 6-byte sequences that represent a single ISO-10646 character. UTF-8 is very easy to transport, has no byte-ordering issues, and is easy to enter (if tedious) if the desired Unicode characters are known. It also mixes very nicely into plain text such as the input files read by USBRC.

I can prepare files in Unicode using Notepad on Windows XP, 2000 or NT. How do I get the data into USBRC?

USBRC can read Windows XP/2000/NT 4.0 Notepad Unicode files directly, in any of the common formats. It will automatically detect these files. A heuristic is used to detect these files, which may cause problems, so you can force USBRC to assume a particular file encoding. SEE COMMAND LINE SYNTAX AT SECTION 4, PAGE 7.

What platforms does USBRC run on?

MCCI has versions of USBRC for Windows (supporting Window 95, 98, 98SE, ME, NT 4.0, Windows 2000, and Windows XP for X86 platforms), NetBSD/386, SunOS 4.1.3, and Solaris 2.5 and later. Other platforms are available by special order.

How do I use USBRC?

USBRC is invoked as a command line utility. You must prepare the input file using a text editor, and then USBRC to convert the file to C. USBRC is designed for integration into a MAKEFILE or BUILD-based programming environment.

2 The Layout of USBRC files

2.1 The Basics

USBRC parses the input files according to some simple rules. The input file is first broken up into **tokens**, which are words, strings and punctuation marks.

USBRC ignores spaces and newlines, except that spaces and newlines always mark the end of a word.

Each of the following punctuation marks is significant to USBRC. The meaning depends on the context.

```
( { } ) . , ; =
```

Each of the following characters begins a longer special token, as defined below:

```
% # "
```

2.1.1 Identifiers

Identifiers are sequences of letters, digits, underscores, minus signs, and other punctuation marks that are not recognizable as numbers. In USBRC input files, you use identifiers to specify names to be used in the C output code, and also (if you choose) to specify strings.

2.1.2 Keywords

Certain identifiers are reserved by USBRC for use as **keywords**:

```
alternate-setting,    application,    bulk,    bus-powered,    class,
configuration,   control,   control-packet-size,   data,   data-pump,
datapump, default, device, device-version, dword, else, end, endpoint,
endpoints, external, full, hidden, high, if, in, Include-File-Name,
Init-Function-Name,     interface,     Interface-Data-Structure-Name,
```

```
interrupt, isochronous, language, low, ma, mux, name, on-the-go, out,
packet-size,    polling-interval,    power,    private-data,    private-
descriptors,    private-descriptor-table,    product-id,    Properties,
protocol, raw, Read-only, rem, remote-wakeup, self-powered, serial-
number, speed, string-index, strings, subclass, three-byte, usb-
resource-file, usb-version, vendor, word
```

These words cannot be used for any other purpose. USBRC ignores uppercase/lowercase distinctions when checking for keywords: "`Bulk`", "`BuLK`", and "`BULK`" are all equivalent ways of writing the keyword "`bulk`".

In addition, some tokens are keywords within certain clauses (but not elsewhere).

2.1.2.1 Special Keywords in the "DataPump" Section

In the "**`DataPump`**" section (section 2.3.8), the following additional keywords are defined:

```
buffer-size, chip-data-structure-name, chip-header-name, chip-name,
config-data-structure-name, endpoint-data-structure-name, endpoint-
mapping, include-file-name, init-function-name, interface-data-
structure-name, number-of-endpoints, setting-data-structure-name,
speeds
```

Otherwise, these keywords have no special meaning.

2.1.2.2 Special Keywords in the "Application" Section

In the "**`application`**" section (section 2.5), the following additional keywords are defined:

```
descriptor, file, filter, function, functions, get-descriptor, global,
header-file, ids, initialization, internal, names, prefix, root, set-
descriptor, static, string, structure, table, type, with, without
```

Otherwise, these keywords have no special meaning.

2.1.2.3 Special Keywords in the "On-The-Go" Section

In the "On-The-Go" section (section 2.5), the following additional keywords are defined:

```
hnp, srp
```

Otherwise, these keywords have no special meaning.

2.1.3 Comments

Comments may be written in two ways. USBRC treats comments the same as spaces.

1. Text included between percent signs ("%") is treated as commentary. The text may include any number of lines. Examples:

```
% this is commentary %
%
   this is a multi-line comment.
%
```

2. Text from "#" to the end of the line is treated as commentary. Examples:

```
# This is a comment
# Note that '%' signs are
# ignored within this kind of comment.

% Similarly, '#' is ignored inside percent signs. %
```

2.1.4 Numbers

Numbers are used to specify many different things, including vendor IDs, product IDs, configuration numbers, endpoint addresses, power consumption, and so on.

USBRC recognizes numbers in several forms.

- Numbers beginning with the digits "1" through "9" are interpreted as decimal numbers.
- Numbers beginning with the strings "0x" or "$" are interpreted as hexadecimal numbers.
- Numbers beginning with "0" are interpreted as octal numbers.

Numbers can have any value from 0 to $2^{32} - 1$ (in hex, 0 to 0xFFFFFFFF). Here are some examples:

```
12    # decimal 12 (binary 1100)
0xC   # decimal 12, in hexadecimal
$C    # decimal 12, in another hexadecimal style
014   # decimal 12, in octal
```

2.1.5 Strings

Strings are written by enclosing characters inside double quotes (""). In USBRC, strings are used for two purposes: to specify string descriptor values, and to select languages for multilingual peripherals.

Internally, USBRC represents all strings in UNICODE. If the input file is encoded in UNICODE or UTF-8, any UNICODE character can be embedded in a string without special handling except for "\", close quote, and newline.

Within strings, the USBRC recognizes the normal C escape sequences:

\a	represents a bell character
\f	represents form-feed
\n	represents newline (line-feed)
\r	represents carriage return
\t	represents a tab
\000 .. \377	represent a single character, encoded in octal. For compatibility with C, USBRC limits the representation to three octal digits.
\x00 .. \xFF	represent a single character, encoded in hexadecimal. For compatibility with C, USBRC limits the representation to two hex digits.
\\	represents a backslash.
\"	represents a quote that is embedded in the string.
end-of-line	Allows a string to be written on multiple lines. The "\" and the new-line character are *not* part of the string.

In addition, USBRC provides the following escape sequence as an extension to the normal C escape sequences.

\{*hex*}	represents an extended (UNICODE or ISO-10646) character. The backslash, left brace and right brace must all be entered literally. *Hex* is a sequence of hex digits, and may represent up to 32-bits of information. If the resulting value is a 16-bit UNICODE character, it will be converted to a single character in the string. If the resulting value can be represented as a two-character (four-byte) extended UNICODE sequence, USBRC will convert it to the appropriate sequence. If the resulting value cannot be represented in UNICODE, USBRC will print an error message.

Here are some examples of strings:

```
"ABC"               # a string consisting of 3 characters (6 bytes)
                    # with values $0041, $0042, $0043

"\n"                # A string consisting of 1 character (2 bytes)
                    # with value $000A

"\x{516c}\x{53f8}"  # a string consisting of 2 characters
                    # (4 bytes).  These are the Chinese characters
                    #  for the word “gongsi”, meaning “company”.

"\x516C"            # a string consistion of 3 characters (6 bytes):
                    # the character $0051 (“Q”), the character
                    # $0036 (“6”), and the character $0043 (“C”).
                    # This is because the C \x sequence scans at
                    # most 2 digits.  This example shows that
                    # “{“ and “}” are very important when entering
                    # UNICODE characters in hex!
"AB\
CD"                 # a string consisting of 4 characters (8 bytes):
                    # $0041, $0042, $0043, and $0044.  The “\”
                    # serves to “escape” the newline.
```

2.1.6 Expression

Expression can be used in the place where numbers and strings are allowed in the input resource file. Expressions must be formed using constants i.e. numbers and strings.

Keywords ‘speed’, ‘low’, ‘full’ and ‘high’ are the only identifiers allowed in relational expression to differentiate high and full speed.

Here are some examples of expression:

```
1 + 2               # An expression involving additive operator with
                    # numbers

"AB" "CD"           # An expression for string concatenation with
                    # strings. Result is equivalent to "ABCD"

"AB" 1              # An expression for string concatenation with a
                    string and a number. Result is equivalent to "AB1"

speed == high       # An expression involving relational operator with
                    keywords as operand
```

2.1.6.1 Operators

The following is the list of operators supported by USBRC.

Operator	Syntax	Description
+	<expression> + <expression>	Adds two number expressions
-	<expression> - <expression>	Subtracts two number expressions
*	<expression> * <expression>	Multiplies two number expressions
/	<expression> / <expression>	Divides two number expressions
%	<expression> % <expression>	Gives remider of a division
	<expression> <expression>	Concatenates two string expressions
!=	<expression> != <expression>	Compares two number expressions for inequality
==	<expression> == <expression>	Compares two number expressions for equality
<	<expression> < <expression>	Compares two number expressions for less than
<=	<expression> <= <expression>	Compares two number expressions for less than or equal
>	<expression> > <expression>	Compares two number expressions for greater than
>=	<expression> >= <expression>	Compares two number expressions for greater than or equal
&	<expression> & <expression>	Bit-wise AND two number expressions
|	<expression> | <expression>	Bit-wise OR two number expressions
^	<expression> ^ <expression>	Bit-wise EX-OR two number expressions
<<	<expression1> << <expression2>	Left shifts <expression1> by <expression2> times
>>	<expression1> >> <expression2>	Right shifts <expression1> by <expression2> times

Operator	Syntax	Description
+	+ <expression>	
-	- <expression>	Negates a number expression
!	! <expression>	Logical NOT of a number expression

~	~ <expression>	Bit-wise NOT of a number expression

Please see section 2.6 'String Expressions in USBRC' for string related expressions.

2.2 The Input Language

The input file consists of the following sections:

- There must be a **`USB-resource-file`** section, which defines the data that is to be emitted to the descriptor file, and also define the layout of the device.
- Optionally, there can be a **`DataPump`** section, which provides information to the resource compiler about the target USB chip. Normally, this information comes from a separate file, provided by MCCI, that matches the target USB chip's miniport driver for the MCCI USB DataPump.
- Optionally, there can be an **`application`** section, which provides information to the resource compiler that customizes the output files to match the application. Much of the information in the **`application`** section can be overridden from the command line. However, for a given application of the DataPump, it is normally more convenient to place the information in the resource file than it is to put it in the make file.

The sections may appear in any order.

2.3 The "USB-resource-file" Section

The **`USB-resource-file`** section has the following structure:

```
USB-resource-file 1.0 =
      {
      USB-Device-Descriptor
      USB-Configuration-Descriptor(s)
      String-Translation-Table(s)
      Private-Descriptor(s)
      Data-Tables(s)
      };
```

The statements in the resource file section may occur in any order.

2.3.1 The "USB-device" Statement

Each device must have one `USB-Device-Descriptor` of the following form:

```
USB-Device
      {
```

```
USB-version major.minor
Class class-number
SubClass subclass-number
Protocol protocol-number
Control-Packet-Size max-bytes,
Vendor USBIF-vendor-ID-number
      [ optional-vendor-name-string-or-ID ]
Product-ID vendors-product-id-number
      [ optional-product-name-string-or-ID ]
Device-Version major.minor
[ Serial-Number string-or-ID ]
} [ optional-label ] ;
```

Each of the parts of a **USB-Device** statement must occur in the order given above. The parts enclosed in square brackets ("[" and "]") are optional.

The three string-or-ID fields, if present, are used in building the string descriptor table. You can write either a string enclosed in quotes, or a USBRC identifier. If you are not worried about translating, you can just write a string enclosed in quotes; see section 5.2, page 16, for an example. If you want to provide support for multiple languages, you can use an identifier, and then provide a string table that specifies the translation. See section 5.3, page 21, for an example.

USBRC automatically assigns the string table indices. USBRC can optionally create a header file containing **#define** commands that define each of the assigned indices. This can be helpful if you are storing some of the descriptors in external EEPROM and need to know what descriptor (in particular) is being assigned.

2.3.2 The "Configuration" Statement

Each device must have one or more USB-Configuration-Descriptor statements. These have the following form.

```
configuration configuration-index
      {
      [ optional-configuration-name string-or-ID ]
      [ bus-powered | self-powered | remote-wakeup ]
      power number [ ma ]
      [ optional-private-descriptor(s) ]
      interface interface-number {
            interface-clause(s) – see below.
            }
      interface interface-number {
            interface-clause(s) – see below.
            }
      } [ optional-label ] ;
```

USBRC translates this statement into a configuration descriptor, followed by the interface descriptors and endpoint descriptors that go with this descriptor.

The fields must appear in the order shown above. Any or all of the flags (**`bus-powered, self-powered, remote-wakeup`**) may appear, in any order before the "**`power`**" keyword.

The power of the device is specified by the "power" keyword. The power is specified in milli-amperes. If the device is not bus-powered, the field should be zero. The keyword "**`ma`**" is optional, and is simply commentary.

Any private descriptors specified are inserted in the resulting configuration descriptor between the basic configuration descriptor and the first interface descriptor.

2.3.3 The "Interface" Statement

Normally, a configuration has at least one interface. Each interface is represented by an **`interface`** statement:

```
interface interface-number {
      [ alternate-setting alternate-setting-code ]
      class interface-class-number
      subclass interface-subclass-number
      protocol interface-protocol-number
      [ name interface-name-string-or-ID ]
      [ optional-private-descriptors ]
      endpoints
            endpoint-statement(s) – see below
      ;

      and optionally…
      [ alternate-setting alternate-setting-code ]
      class interface-class-number
      … and so forth
      ;
      }
```

USBRC translates each **`interface`** statement into one or more interface descriptors, with associated endpoints. One interface descriptor is produced for each alternate setting for the interface. If the `alternate-setting-code` is zero, the "**`alternate-setting`** `alternate-setting-code`" phrase can be omitted.

Because some interface settings might not have any valid endpoints associated with them, the **`endpoints`** statement is optional.

2.3.4 The "Endpoints" Statement

The endpoints of an interface/alternate-setting are represented by the **`endpoints`** statement.

```
endpoints
      endpoint-specification(s)
```

Each `endpoint-specification` has the following form:

```
[ control | isochronous | bulk | interrupt ]
      [ in | out ]
      endpoint-address-number
      packet-size maximum-size-in-bytes
      [ polling-interval polling-interval-in-milliseconds ]
      [ data { raw-data-sequence } ]
      [ optional-private-descriptors ]
```

Not all of the combinations of the above options make sense. **Interrupt** endpoints must always be **in**. The direction of **control** endpoints is ignored. USBRC verifies that the specified **packet-size** makes sense for the endpoint type, according to the USB specification. **Polling-interval** should only be specified for **isochronous** and **interrupt** endpoints.

The endpoint-address-number must be between 1 and 15, inclusive. USBRC automatically generates the appropriate endpoint descriptor based on the type, direction, packet-size, and polling-interval.

If the **data** clause is present, then the data bytes in the raw-data-sequence are appended to the generated endpoint descriptor. This is useful for generating Audio-class endpoint descriptors.

If the optional-private-descriptors are present, then the specified descriptors are inserted into the configuration bundle after the specified endpoint descriptor.

2.3.5 The "Strings" Statement

The **strings** statement is used to provide translations for USB device strings in multiple languages.

```
strings
      {
      optional-property-list
      language-spec-list {
            string-or-ID = string-in-these-languages ;
            string-or-ID = string-in-these-languages ;
            . . .
            }
      language-spec-list
            {
            string-or-ID = string-in-these-languages ;
            string-or-ID = string-in-these-languages ;
            . . .
            }
      } [ optional-label ] ;
```

USBRC automatically generates the string descriptors and data structures needed to represent the strings you specify, including string descriptor 0, which returns a list of language IDs.

Some string descriptors may need to be stored in EEROM or other storage external to the tables generated by the resource compiler. Usually, only a few descriptors need to be stored externally. You identify those descriptors by providing one or more optional-property-list statements. Each statement has the form:

```
properties external ID [ , ID . . . ] ;
properties external read-only ID [ , ID . . . ] ;
```

Notice that the permitted set of properties is either "**external**" or "**external read-only**".

Specifying that a string is external *does not* change how USBRC records data about the string. Normally, you'll provide the default or initial values for the strings as part of the resource file, and you'll arrange to supersede these values with externally-stored values. If you are using the MCCI USB DataPump, most of this process is automatically handled for you.

Since USBRC assigns the string indices automatically, you may need a way to associate the string indices with specific string descriptors. (For example, the serial number might be computed based on the Ethernet node ID, which might be stored in a network controller chip.) USBRC can generate an additional header file (the *string index header file*), which automatically defines symbolic names for each of the **external** string descriptor IDs. The names are based on the IDs you use in the USB resource file.[1]

Each language-spec-list specifies one or more language codes (as defined by the USB specification) for which you want to provide translations. Each entry in the list is separated by commas. You can use the following things in the list:

```
default | language language-id-number | language language-name-string
```

If you use language-name-string, you can specify the language directly. The language name *must* be quoted. Language name comparisons are not case-sensitive.

The known languages are as defined in the specification "Universal Serial Bus Language Identifiers (LANGIDs), 3/29/00, Version 1.0", available to USB-IF members through the website http://www.usb.org/.

[1] You can also arrange for these names to be defined in the same header file that defines the structure of your device for the DataPump.

```
"Afrikaans"
"Albanian"
"Arabic"
"Arabic (Algeria)"
"Arabic (Bahrain)"
"Arabic (Egypt)"
"Arabic (Iraq)"
"Arabic (Jordan)"
"Arabic (Kuwait)"
"Arabic (Lebanon)"
"Arabic (Libya)"
"Arabic (Morocco)"
"Arabic (Oman)"
"Arabic (Qatar)"
"Arabic (Saudi Arabia)"
"Arabic (Syria)"
"Arabic (Tunisia)"
"Arabic (United Arab Emirates)"
"Arabic (Yemen)"
"Assamese"
"Azeri (Cyrillic)"
"Azeri (Latin)"
"Basque"
"Belarusian"
"Bengali"
"Bulgarian"
"Burmese"
"Catalan"
"Chinese (Hong Kong)"
"Chinese (Hong Kong SAR)"
"Chinese (Macau SAR)"
"Chinese (PRC)"
"Chinese (Singapore)"
"Chinese (Taiwan)"
"Chinese (simplified)"
"Chinese (traditional)"
"Croatian"
"Czech"
"Danish"
"Dutch"
"Dutch (Belgium)"
"Dutch (Netherlands)"
"English"
"English (Australia)"
"English (Belize)"
"English (Canada)"
"English (Caribbean)"
"English (Eire)"
"English (Ireland)"
"English (Jamaica)"
"English (New Zealand)"
"English (Philippines)"
"English (South Africa)"
"English (Trinidad)"
"English (UK)"
"English (United Kingdom)"
"English (United States)"
"English (USA)"
"English (Zimbabwe)
"Estonian"
"Faeroese"
"Farsi"
"Finnish"
"French"
"French (Belgium)"
"French (Canada)"
"French (France)"
"French (Luxembourg)"
"French (standard)"
"French (Switzerland)"
"German"
"German (Austria)"
"German (Germany)"
"German (Liechtenstein)"
"German (Luxembourg)"
"German (standard)"
"German (Switzerland)"
"Greek"
"Gujarati"
"Hebrew"
"HID (Usage Data Descriptor)"
"HID (Vendor Defined 1)"
"HID (Vendor Defined 2)"
"HID (Vendor Defined 3)"
"HID (Vendor Defined 4)"
"Hindi"
"Hungarian"
"Icelandic"
"Indonesian"
"Italian"
"Italian (Italy)"
"Italian (standard)"
"Italian (Switzerland)"
"Japanese"
"Kannada"
"Kashmiri (India)"
"Konkani"
"Korean"
"Korean (Extended Wansung)"
"Korean (Johab)"
"Latvian"
"Lithuanian"
"Lithuanian (classic)"
"Macedonian"
"Malay (Brunei Darussalam)"
"Malay (Malaysian)"
"Malayalam"
"Manipuri"
"Marathi"
"Nepali (India)"
"Norwegian (Bokmal)"
"Norwegian (Nynorsk)"
"Oriya"
"Polish"
"Portuguese"
"Portuguese (Brazil)"
"Portuguese (Portugal)"
"Portugese (standard)"
"Romanian"
"Russian"
"Serbian (Cyrillic)"
"Serbian (Latin)"
"Sindhi"
"Slovak"
"Slovenian"
"Spanish"
"Spanish (Argentina)"
"Spanish (Bolivia)"
"Spanish (Castilian)"
"Spanish (Chile)"
"Spanish (Colombia)"
"Spanish (Costa Rica)"
"Spanish (Dominican Republic)"
"Spanish (Ecuador)"
"Spanish (El Salvador)"
"Spanish (Guatemala)"
"Spanish (Honduras)"
"Spanish (Mexican)"
"Spanish (Modern)"
"Spanish (modern sort)"
"Spanish (Nicaragua)"
"Spanish (Panama)"
"Spanish (Paraguay)"
"Spanish (Peru)"
"Spanish (Puerto Rico)"
"Spanish (traditional sort)"
"Spanish (Uruguay)"
"Spanish (Venezuela)"
"Sutu"
"Swahili (Kenya)"
"Swedish"
"Swedish (Finland)"
"Swedish (Sweden)"
"Tamil"
"Tatar (Tatarstan)"
"Telugu"
"Thai"
"Turkish"
"Ukrainian"
"Urdu (Pakistan)"
"Urdu (India)"
"Uzbek (Cyrillic)"
"Uzbek (Latin)"
"Vietnamese"
```

Each string-or-ID is one of the strings or identifiers previously mentioned in a **USB-Device**, **configuration**, or **interface** statement.

Each string-in-these-languages gives the translation (as a UNICODE string). To specify that the translation should be the same as the default language, you can use the keyword **default** and the syntax:

```
string-or-ID = default ;
```

USBRC will notice if a given string is used more than once, and will only generate one copy of that string descriptor. The string descriptor will be used as many times as needed in the generated data structures.

2.3.6 The "Private-Descriptor" Statement

For special applications, USBRC allows you to create private descriptors. The syntax of the **private-descriptor** statement is:

```
private-descriptor [ optional-label ]
      {
      [ Raw-Descriptor |
        Configuration |
        Interface |
        Endpoint |
        Device ] . . .
      };
```

Configuration, Interface, Endpoint, and Device statements are as given above; however, the Interface and Endpoint statements generate only the corresponding interface and endpoint descriptors.

Raw-Descriptor is used to generate descriptors with arbitrary contents. The syntax is:

```
raw { code-byte data };
```

code-byte is the descriptor code; data specifies the byte data to be embedded in the descriptor as a list of bytes or strings. USBRC automatically calculates and inserts the descriptor length for you. See section 2.6, page 5, for more details on the operators that can be used to construct the data.

The label is used to label the generated table of descriptors, if **private-descriptor** is used at the top level of the USB resource file; otherwise it is ignored.

A simple example is:

```
private-descriptor
      {
      # generate a RAW descriptor of type 0x55, with a body consisting
      # of bytes 3, 5, 7, 0, and the unicode version of the string.
      # The 7,0 comes from the "word" operator.
```

```
        raw { 0x55 3 5 word(7) "This string will be stored as Unicode" };
        }
```

Another example:

```
    private-descriptors
        {
        %
          generate a raw descriptor entry of type 0xDB, length 7,
          containing data 0x06 0xDB 0xA0 0x86 01 00, and the string index
          for the tag "SERIALNUMBER".  100000 is 0x186A0, hence the
          sequence 0xA0, 0x86, 1, 0.
        %
        raw { 0xDB dword(100000) string-index SERIALNUMBER };
        }
```

Private-descriptors are most frequently used for inserting additional (newly standardized or proposed standard) descriptors that the USBRC compiler doesn't yet directly support.

2.3.7 The "Data" Statement

Data allows you to use USBRC to generate arbitrary data in a portable way, using any of the types supported by USBRC.

The syntax is:

```
    data label
        {
        byte-or-string-data
        };
```

2.3.8 The "On-The-Go" Statement

Each device must have one On-The-Go descriptor of the following form:

```
    on-the-go [srp] [hnp] ;
```

Each of the parts of an **On-The-Go** statement must occur as given above. The parts enclosed in square brackets ("[" and "]") are optional.

2.4 The "DataPump" Section

The DataPump-section is optional. It supplies information that is used for building the initialization functions and data structures required by the MCCI USB DataPump. If you need to create the c file or header file for use with the DataPump, you must add this section to the input URC file, or add the name of the file containing this information to the USBRC command line. Because USBRC will accept multiple file names on its command line, it is normally more convenient to keep this information in a separate, chip-specific file, which can be selected using

the "-chipinfo" switch. Normally this file is supplied by MCCI, as a separate URC file for each supported chip.

The keyword "Data-Pump" is also recognized, for backwards compatibility with older versions of USBRC.

The section has the following form:

```
DataPump
      {
      Chip-Name   chip-name-string
      Include-File-Name  include-file-name-string
      Chip-Header-Name  chip-header-name-string
      [Chip-Data-Structure-Name  data-structure-name-string]
      [Init-Function-Name   init-function-name-string]
      Endpoint-Data-Structure-Name endpoint-name-string
      [Config-Data-Structure-Name  config-name-string]
      [Interface-Data-Structure-Name interface-name-string]
      [Setting-Data-Structure-Name setting-name-string]
      Endpoint-Number   numEndpoint-number
      Speeds low|full|high[ low|full|high…]
      EndpointMappingExpr
      };
```

Each of the parts of a **DataPump** section must occur in the order given above. The parts enclosed in square brackets ("[" and "]") are optional. You must write each of the string fields, if present, as a string enclosed in quotes.

Include-File-Name gives the name of the file which has the definitions for the use of the chip-specific kernel. For example, it could be "u820kern.h". This must match the name that is used by the DataPump chip driver.

Chip-Header-Name gives the name to be used as the chip header Data Structure. For example, we could have "UDEV620_HDR", "UDEV820_HDR", etc. The name chosen must match the name used by the DataPump chip driver.

Chip-Data-Structure-Name gives the name of the chip-level Data Structure. It must match the name that is used for this purpose by the DataPump chip driver. For example, we can have "UDEV620", "UDEV820", etc. It is optional, and the default name is UDEV{Chip-Name}. For example, suppose the chip name is xyz, then the default name would be: UDEVxyz.

Endpoint-Data-Structure-Name gives the name of the chip-specific endpoint data structure. It must match the name that is used for this purpose by the DataPump chip driver. For example, we can have "UEP620", "UEP820", etc. This field is optional, and the default name is UEP{Chip-Name}. For example, suppose the chip name is xyz, then the default name would be: UEPxyz.

Init-Function-Name gives the name of the function that initializes the chip-level data structure. It is optional, and the default name is: "UsbPumpInit".

Config-Data-Structure-Name gives the name of the config data structure. It is optional, and the default name is: `"UCONFIG"`. You should not change the value, as it must match the name used by the DataPump common code.

Interface-Data-Structure-Name gives the name of the interface set data structure. This is the data structure that collects all alternate settings for a given interface. This field is optional, and the default is: `"UINTERFACESETS"`. You must not change the value, as it must match the name used by the DataPump common code.

Setting-Data-Structure-Name gives the name of the interface data structure. The interface data structure models a single alternate setting of an interface. This field is optional and the default is: `"UINTERFACE"`. You must not change the value, as it must match the name used by the DataPump common code.

Endpoint_Number gives the total number of the physical endpoints that are defined by the hardware.

Speeds gives the speeds that are supported by this device among low, full and high.

`EndpointMappingExpr` provides the mapping from endpoint address to endpoint index. The requirement is that for each endpoint address, there is only one corresponding endpoint index.

The **`EndpointMappingExpr`** has the following syntax:

1. if <expr> <mapping-set> else <mapping-set>
2. <mapping-set>

The mapping set has the following syntax

Endpoint-Mapping
{
(endpoint_address**,** endpoint_index**)…**
}

The meaning of the "endpoint index" is defined by the chip-specific code that uses it.

Please see section 5.5, page 32, for an example.

2.5 The "Application" Section

Each input file may have an **`application`** section, of the following form:

```
application
      {
      descriptor table {
            functions {
                  [ with | without ] string filter table;
                  get-descriptor get-fn-string ;
                  set-descriptor set-fn-string ;
                  };
            internal names [ static | global ] ;
```

```
        root name root-table-name ;
        };
header-file {
        file name header-file-name ;
        [ with | without ] string ids;
        };
initialization {
        file name init-code-file-name ;
        function name init-fn-name ;
        };
strings {
        file name string-id-file-name ;
        prefix prefix-string ;
        };
structure {
        type name structure-name ;
        };
};
```

All of the subsections of the **`application`** section are optional, and may appear in any order. For ease of reference, the above table is sorted in alphabetical order.

All of the variable fields in the application structure should be specified as strings. The string fields are used by USBRC for the purposes given in Table 1.

Table 1. String Values in Application Section

get-fn-string	The name of the function that is to be called to filter get-descriptor operations.
header-file-name	The reference name for the data pump header file, which will contain the structural definition for the device described by this resource file.
init-code-file-name	The reference name for the initialization code file, which will contain the C function that initializes the data structures for this device.
init-fn-name	The name to be given to the initialization function.
prefix-string	The string that is to be prefixed to the string-index symbolic definitions.
root-table-name	The name to be given to the root descriptor table.
set-fn-string	The name of the function that is to be called to filter set-descriptor operations.
string-id-file-name	The reference name for the string-id header file, which will contain the symbolic names of the exported string IDs.
structure-name	The typedef name for the structure that models this USB device

In addition, several Boolean options are provided.

`descriptor table … functions …` [**`with`** | **`without`**] **`string filter table`** controls whether USBRC generates additional bit-maps for the root descriptor table. **`With string`**

`filter table` causes USBRC to generate two bit-maps: one for the string IDs that were declared to be **`external`**, and another for the string IDs that were declared to be **`read-only.`**

`descriptor table … names` [**`static`** | **`global`**] controls whether USBRC uses global or static definitions to generate the resource table. **`Global`** (the default) causes USBRC to use global definitions. **`Static`** causes USBRC to mark all global data cells in the data file as static, except for the root table.

`header file … with string ids` causes USBRC to generate string ID definitions into the DataPump header file. **`header file … without string ids`** causes USBRC to omit the string ID definitions from the DataPump header file. (**`Without`** is the default.) The creation of a separate string ID file is not affected by this option. String ID definitions can be sent to either place, or to both places.

Please see section 5.6, page 33 for an example of a completed **application** section.

2.6 String Expressions in USBRC

USBRC can parse a limited set of expression operators when building string and private descriptor values. The following are recognized:

`String`	Generates a UNICODE string, with each character stored in low/high order.
`Number`	Generates a single byte of data.
`Word(` `number` **`)`**	Generates two bytes of data, the number stored in low/high order.
`three-byte(` `number` **`)`**	Generates three bytes of data, representing the number stored in low/high order.
`dword(` `number` **`)`**	Generates four bytes of data, the number stored in low/high order.
`language` `language-name-string`	Generates two bytes of data, the language code corresponding to the given language name.
`(` `string-value` `string-value` … **`)`**	Concatenates each of the specified values together to form a single string value.
`string-index` `ID`	Generates a single byte, containing the string index assigned to the specified identifier.
`string-index` `literal-string-value`	Generates a single byte, containing the string

	index assigned to the specified string value.
`string-index (` `string-expression` **`)`**	Generates a single byte, containing the string index assigned to the specified string value.
`string-expression string-expression`	Concatenates two string expressions

Please see section 5.7, page 34 for a sample device description that uses these constructs to add several USB Communication Device Class descriptors to a configuration descriptor.

3 Using the output

Please refer to the header file "usbrctab.h", which is included in the USBRC distribution, for information on how to define the macros that USBRC expects.

4 Command Line Reference

NAME

usbrc- Universal Serial Bus descriptor and string resource compiler.

SYNOPSIS

usbrc *options* input-file(s)

Where *options* are:

- Input file control: this group tells USBRC how to process the input files.
  [**-e** input-file-encoding]

- Output file control: this group tells USBRC where to put its output, and what to call the files it produces.
  [**-Ho** header-output-file]
  [**-Io** initialization-output-file]
  [**-o** table-output-file]
  [**-Xo** string-ID-output-file]

- Compilation Overrides: this group allows you to override many of the names compiled into the files produced by USBRC. All of this information is normally placed in the "**application"** section of the input files, but it can be convenient to be able to specify this information on the command line as well.
  [**-i** header-to-include-in-table]
  [**-Fn** initialization-function-name]
  [**-Hn** header-reference-name]
  [**-In** init-file-reference-name]
  [**-Rn** name-of-descriptor-table]

[**-Rs**]
[**-Sn** name of top-level structure]
[**-Xn** string-ID-file-reference-name]
[**-Xp** prefix-for-string-ids]

- Miscellaneous:
  [**-chipinfo** filename]
  [**-dpapi** bitmask]
  [**-type** output-type]
  [**-v**]
  [**-Werror**]

DESCRIPTION

USBRC is a utility that allows USB developers to create and manipulate USB device descriptors in a high level format. USBRC reads a device description from the input files (conventionally of the type "**.urc**"), and produces equivalent tables as an ANSI C module.

When used with the MCCI USB DataPump, USBRC can optionally generate a header file that describes the device to the DataPump, an initialization function that will set up the data structure at run-time, and a header file containing string index information.

Therefore, USBRC can produce up to four files in a single compilation:

1. The descriptor tables (the "table file"). This file is written to STDOUT, or to the file specified by the "**-o**" option. It is always produced.

2. A header file describing the logical structure of the device (the "data pump header file"). This file is written to the file specified by the **-Ho** option, and is only written if **-Ho** is specified.

3. A C file containing a function that will initialize the data structures appropriately (the "data pump initialization file"). This file is written to the file specified by the **-Io** option, and is only written if **-Io** is specified.

4. A header file containing symbols equated to the indices of string descriptors that were declared as being externally stored. This simplifies the coding of the functions that provide these descriptors. This file is only written if "**-Xo**" is specified.

USBRC operates by parsing each of the input files in turn, treating them as a single logical input file. After the input has been parsed, it produces the requested output files.

The data pump header file and the data pump initialization file refer to files and functions by name. For example, the data pump initialization file **`#include`**s the data pump header file. For convenience, you specify the reference names *that are compiled into the code* separately from the names *that are used for the output files.* The options -Ho and -Io specify only the names of the output files. Reference names can be obtained from the "**application**" statement or from the command line.

The following options are available**:**

-chipinfo filename Specifies the name and location of the "chip information" file. The file named by this switch is read after all the other input files have been read. Specifying the chip information file as part of the USBRC options can make it easier to build Makefiles.

-dpapi bitmask Specifies MCCI USB DataPump API options to be used when compiling the resource file. The bitmask is specified as a number, in decimal, octal or hexadecimal (as in C). Within the mask, each bit defines a separate option. If omitted, the generated code is compatible with DataPump API versions through V1.78. The defined bits are:

-dpapi 1 USBRC generates extra code required starting with V1.79a, which places pointers to the relevant descriptors in the initialized UDEVICE, UCONFIG, UINTERFACE and UENDPOINT structures.

-e encoding Input file text encoding method. The possible selections are:

-e auto USBRC automatically detects the encoding scheme. This is the default.

-e ascii The input file is in ASCII; no multi-byte characters are embedded in the file.

-e utf8 The input file is encoded in UTF-8. This encoding is the same as **ascii** for normal text. If the file begins with the character sequence 0xEF, 0xBB, 0xBF, USBRC automatically detects that the input file is UTF-8 text. Otherwise, USBRC will default to reading ASCII. If you have UTF-8 text that does not begin with the signature, you will need to use this switch to force USBRC to decode the input as being UTF-8.

-e unicode The input file is in UNICODE, in Windows NT/x86 byte order. Normally, you don't need to specify this, because USBRC will automatically detect this kind of file.

-e unicode/big The input file is in UNICODE, with the most significant byte first. Again, normally this encoding is automatically detected.

-e ucs4/little The input file is in ISO-10646 4-byte characters, with the least-significant byte of each character first.

-e ucs4/big The input file is in ISO-10646 4-byte characters, with the most-significant byte first.

-Fn function Specifies the name of the initialization function that is to be generated. The default is `UsbPumpInit`; however, this is usually overridden by the "**DataPump**" section or the "**application**" section of the input file. This is used to create both the initialization .c file and the data pump header file.

-Hn file.h Specifies the reference name of the data pump header file that is compiled into the output files.

-Ho file.h Requests that a data pump header file be generated, and specifies the name of the file. If the file name is specified as "-", USBRC will write its output to STDOUT.

-i header Controls the name of the header file that is **`#include`**d by the table file. If **-i** is not specified, then the value given for **-Hn** is used; if neither **-i** nor **-Hn** is given, USBRC will use `#include "usbrctab.h"`.

You may want to use a different file in the table file for some applications. Using **-i**, you can override the default in one of two ways.

-i file.h USBRC will insert `#include "file.h"` at the appropriate place in the table file.

-i "" USBRC will not insert any `#include` command in the table file. This may be appropriate if you want to generate the table file as a ".h" file rather than a ".c" file.

-In file.c Specifies the reference name of the data pump initialization file that is compiled into the output files.

-Io file.c C file which contains the initialization function. If the file name is specified as "-", USBRC will write its output to STDOUT.

-o table-file Designates the output file for the table file. Normally, USBRC writes its output to STDOUT. Using "**-o**", you can send the output directly to a file without redirection.

-Rn name-of-descriptor-table
Designates the name to be used for the descriptor table, and for the subsidiary tables produced by this compilation. Normally, USBRC assigns default names based on information assigned in the "application" section of the input file.

-Rs USBRC will declare all tables except the "topmost" table as being static. Normally, or with "**-noRs**", all tables are global. This overrides any information given in the "**application**" section of the input file.

-Sn structure Name of top-level data pump structure.

-type type Specifies the format of the output file. Default is "**-type c-source**". In this version of USBRC, this is the only output format available.

-v Display copyright and version information (e.g. Version 2.22) at startup. The default is to run quietly.

-Werror Treats warning messages as errors. By default, warnings are displayed but do not affect the exit status of the program. If **-Werror** is specified, USBRC will treat warnings as errors, and will exit with error status if any warnings or errors are detected.

-Xh Requests that string IDs be defined, and that the definitions be included in the Data Pump header file. Default is not to include this information. Conflicts with **-Xo**.

-Xn name-to-compile-into-string-id-file
If a separate string ID file is selected, this name will be compiled into the header for that file.

-Xo string-id-output-file
Specifies that a separate string ID file is to be created. If the file name is "-", the file is written to STDOUT.

-Xp prefix-string
The specified prefix will be added to each generated string ID symbol.

EXIT STATUS

USBRC exits with status 0 if the input file was successfully translated, or was translated with warnings. It exits with status 1 if errors were detected. (If **-Werror** is specified, then USBRC will exit with status 1 if any warning messages are detected.)

ENVIRONMENT VARIABLES

USBRC examines the value of the environment variable USBRCFLAGS for additional options. These options are scanned before the command line options are scanned.

NOTES

USBRC does not remove the output files created by the "-o", "-Ho" or "-Io" options if it exits with errors.

EXAMPLES

Assume that you have a URC file named "mydevice.urc" in the current directory, and a chip definition URC file named "uss820.urc", also in the current directory.

To create a data pump header file named "mydevice.h", you can call the resource compiler this way:

```
usbrc.exe -o nul –chipinfo uss820.urc -Fn chip_app_init -Hn mydevice.h \
      -Ho mydevice.h -Sn UDEVCHIP_APP mydevice.urc
```

The "-o nul" option (on Windows) causes the descriptor table information to be discarded.

To create a data pump initialization c file named "mydevice.c" for the same chip application, you can call the resource compiler this way:

```
usbrc.exe –chipinfo uss820.urc -Fn chip_app_init -Hn mydevice.h -In mydevice.c
\
      -Io - -Sn UDEVCHIP_APP -o - mydevice.urc >mydevice.c
```

(This example generates a file that contains both the initialization code and the descriptor tables. "-o -" redirects the descriptor table output to STDOUT, and "-Io -" redirects the initialization function output to STDOUT. ">mydevice.c" sends the combined output to the file "mydevice.c".)

To create the table, header, and initialization files at the same time:

```
usbrc.exe –chipinfo uss820.urc -Fn chip_app_init -Hn mydevice.h -Ho mydevice.h \
      -In mydevice.c -Io - -Sn UDEVCHIP_APP -o - mydevice.urc >mydevice.c
```

5 Example resource description files

Following are example resource description files. These files use the syntax outlined above; here, however, the keywords are not all lower case. For each example there is a corresponding descriptor table output file, showing the expected results of a successful compilation.

No examples are given of the corresponding data pump header and initialization code files.

5.1 A Simple Device

Input file:

```
# ex1.urc
#
# USB-resource-file with device descriptor and
# config descriptor and interface descriptor, with
# first setting (but nothing else)
#
# Test-Options:
#


USB-resource-file 1.00 =
      {
      Device  {
```

Commentary. Since this is commented it is not analyzed by USBRC.

The remainder of the material in this column are referenced to the Standard USB Descriptor Definition listed in 9.6 of the Universal Serial Bus Specifications Revision 1.00

The material in this section is placed into the device descriptor.

`USB-version 1.00`	Specifies the version of the USB specification with which this device claims compliance.
`Class 0`	Specifies the class of the device. bDeviceClass code assigned by the USB standard or the USB-IF.
`SubClass 0`	See USB Class Code Manual for required class and subclass descriptors for values between 1 and 0xFE.
`Protocol 0`	Protocol is qualified by the device class and subclass and is assigned by USB. Values are available from the USB-IF.
`Control-Packet-Size 8`	Packet size is hardware specific and determined by the interface hardware. For practical reasons, this should always be set to 8.
`Vendor 0x040E % string name omitted %`	Vendor number is supplied by USBIF. Vendor ID 0x040E is MCCI's vendor ID. Please change this to your own company's ID! Feel free to contact MCCI for assistance in getting an ID if you're not sure how to go about getting one.
`Product-ID 0x1234 % name omitted %`	Product ID is determined by the vendor.
`Device-Version 2.05`	Device version is determined by the vendor. It must be between 0.0 and 99.99.
`% serial-number omitted %`	In this example, we have not provided a serial number clause.
`} %no label for device% ;`	As the comment indicates, we have not named the device descriptor. Naming of device descriptors is not useful in this version of USBRC.
`#### here's the first (and only) configuration.`	bLength, bDescription, wTotalLength and bNumberInterfaces are all automatically computed from user data.

```
Configuration 1
```

bConfigurationValue is specified by the user. Since the device has only one configuration, and since configurations are numbered starting at one, this is configuration 1.

```
{
    % configuration name would go here %
```

We have not provided a configuration name in this example.

```
    Self-Powered Remote-Wakeup Bus-Powered
        %the flags%
```

bmAttributes is determined automatically from the combination of flags specified. Here, we have specified all the flags.

```
    Power 10 mA
```

bMaxPower is determined automatically from the value specified.

```
% private descriptors would go here %
```

For example, the Microsoft power-management descriptors could be coded here.

```
# Next we have the (only) interface
    interface 0
        {
```

Interfaces are specified following after the information that is common to the configuration. In this case we have a single interface; since interfaces are numbered starting at zero, this is interface 0.

```
% Alternate-Setting 0 %
```

Each interface has an alternate setting value. By default, the alternate-setting is zero, so USBRC allows you to omit it in this case.

```
            Class 1
            Subclass 2
            Protocol 3
```

Each interface (and alternate setting) has its own device class, subclass, and protocol code. In this case, we are using arbitrary values.

```
            % name {stringid} %
```

Each interface (and alternate setting) can have its own string identification. In this case, we have omitted a name.

```
            % private descriptors %
```

The private descriptors placed here will be placed in the configuration descriptor between this interface descriptor and the associated endpoints. We have not used any in this example.

```
                % endpoints for this setting %

                endpoints
                    %only one:%
                    interrupt

                    in

                    %address% 0x01

                    packet-size 1


                    polling-interval 1

                %more endpoints would be here%
                ;
            }
        } %no label on the configuration descriptor% ;
    } %no label on the overall resource file% ;
% end of file %
```

Finally, we can specify zero or more endpoints associated with this setting. In this example, the device has a single interrupt endpoint.

The type of endpoint.

The direction – *into* the host.

The endpoint address.

The maximum packet size – only one byte.

For interrupt endpoints, you must specify how often the host should poll, in milliseconds.

Output file (generated with `-dpapi 0`):

```
/* <<commentary omitted for brevity>> */

#include "usbrctab.h"

/* beginning of resource data */
RESCHAR gk_UsbResourceData[] =
      {
      18, 1, 0, 1, 0, 0, 0, 8,
      14, 4, 52, 18, 5, 2, 0, 0,
      0, 1, 9, 2, 25, 0, 1, 1,
      0, 224, 5, 9, 4, 0, 0, 1,
      1, 2, 3, 0, 7, 5, 129, 3,
      1, 0, 1
      };

/* beginning of descriptor table */
RESPTRS gk_UsbResourceTable_1[] =
      {
      RESPTRVAL(gk_UsbResourceData, 0)
      };  /* end of gk_UsbResourceTable_1 */
```

Device descriptor data

Configuration 1 descriptor data

Interface 0 descriptor data

Endpoint descriptor data

Table of pointers to device descriptors.

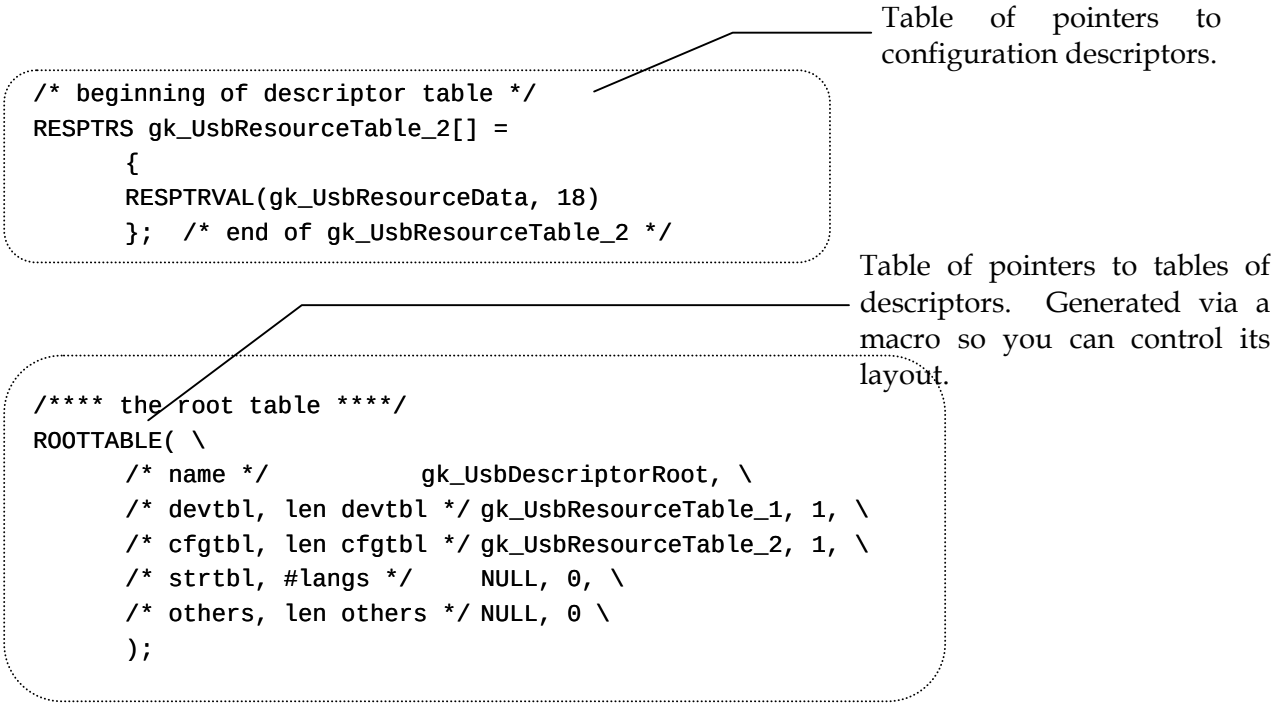
Table of pointers to configuration descriptors.
Table of pointers to tables of descriptors. Generated via a macro so you can control its layout.

```
/* beginning of descriptor table */
RESPTRS gk_UsbResourceTable_2[] =
      {
      RESPTRVAL(gk_UsbResourceData, 18)
      };  /* end of gk_UsbResourceTable_2 */

/**** the root table ****/
ROOTTABLE( \
      /* name */          gk_UsbDescriptorRoot, \
      /* devtbl, len devtbl */ gk_UsbResourceTable_1, 1, \
      /* cfgtbl, len cfgtbl */ gk_UsbResourceTable_2, 1, \
      /* strtbl, #langs */     NULL, 0, \
      /* others, len others */ NULL, 0 \
      );

/***** end of generated data *****/
```

5.2 A Device with Multiple Interfaces

This example shows how to prepare the descriptors for a device with multiple interfaces.

Input file:

```
# ex2.urc
#
# USB resource file with device descriptor,
# config descriptor and interface descriptors,
# as follows:
#
#     ifc 0 = 1 interrupt endpoint
#     ifc 1/alt setting 0 = 2 bulk endpoints (i/o)
#     ifc 1/alt setting 1 = 2 iso endpoints (i/o)
#     ifc 2/alt setting 0 = 2 bulk endpoints (i/o)
#     ifc 2/alt setting 1 = 2 iso endpoints (i/o)
#     ifc 3/alt setting 0 = 2 bulk endpoints (i/o)
#     ifc 3/alt setting 1 = 2 iso endpoints (i/o)
#
# This example also includes string descriptors in the
# default language.

USB-resource-file 1.00 =
      {
```

```
Device  {
    USB-version 1.00
    Class 0xFF  #vendor-specific
    SubClass 0xFF   #serial device
    Protocol 0x0#none specified.
    Control-Packet-Size 8
    Vendor 0x40E "Moore Computer Consultants, Inc"
    Product-ID 0xE101 "USB Demo Firmware"
    Device-Version 1.00
    % serial-number omitted %
    } %no device-descriptor label % ;

#### here's the first (and only) configuration
Configuration 1
    {
    "Default configuration"
    Remote-Wakeup Self-Powered   %the flags%
    Power 0 mA
    % private descriptors would go here %

    # interface 0 has the interrupt endpoint;
    # it has only one alternate setting.
    interface 0
        {
        % Alternate-Setting 0 %
            Class 0     #none defined.
            Subclass 0  %no subclass%
            Protocol 0xFF   %vendor-specific%
            name "Interrupt endpoint"
            % private descriptors %

            endpoints
                %only one:%
                interrupt
                in %address% 0x01
                packet-size 1
                polling-interval 1
            ;
        } %end interface 0%

    # each of interfaces 1, 2, and 3 can be configured
    # independently for bulk or isochronous data transfer.
    interface 1
        {
        # the bulk setting
        alternate-setting 0
            class 0     #none
            subclass 0x0#none
            protocol 0xFF   #vendor-specific.
```

```
            name "Bulk Data"
            endpoints
                bulk out 2 packet-size 16
                bulk in 3 packet-size 16
            ;
        # the isochronous setting
        alternate-setting 1
            class 0     #none
            subclass 0  #none
            protocol 0xFF   #vendor-specific.
            name "ISO Data"
            endpoints
                isochronous out 2 packet-size 16
                    polling-interval 1
                isochronous in 3 packet-size 16
                    polling-interval 1
            ;
        } %end interface 1%

    % interface 2 is used for sending data %
    interface 2
        {
        alternate-setting 0
            class 0     #none
            subclass 0x0#none
            protocol 0xFF   #vendor-specific.
            name "Bulk Data"
            endpoints
                bulk out 4 packet-size 16
                bulk in 5 packet-size 16
            ;
        alternate-setting 1
            class 0     #none
            subclass 0  #none
            protocol 0xFF   #vendor-specific.
            name "ISO Data"
            endpoints
                isochronous out 4 packet-size 16
                    polling-interval 1
                isochronous in 5 packet-size 16
                    polling-interval 1
            ;
        } %end interface 2%

    % interface 3 is used for sending data %
    interface 3
        {
        alternate-setting 0
            class 0     #none
```

```
                    subclass 0x0 #none
                    protocol 0xFF   #vendor-specific.
                    name "Bulk Data"
                    endpoints
                        bulk out 6 packet-size 16
                        bulk in 7 packet-size 16
                    ;
                alternate-setting 1
                    class 0     #none
                    subclass 0  #none
                    protocol 0xFF   #vendor-specific.
                    name "ISO Data"
                    endpoints
                        isochronous out 6 packet-size 16
                            polling-interval 1
                        isochronous in 7 packet-size 16
                            polling-interval 1
                    ;
                } %end interface 3%

            } %end configuration; no label% ;
        } %end resource data; no label% ;

    % end of file %
```

Output file:

```
    /* <<Commentary omitted for brevity>> */
    #include "usbrctab.h"

    /* beginning of resource data – some linebreaks removed for brevity */
    RESCHAR gk_UsbResourceData[] =
          {
          18, 1, 0, 1, 255, 255, 0, 8, 14, 4, 1, 225, 0, 1, 1, 2,
          0, 1, 9, 2, 163, 0, 4, 1, 3, 96, 0, 9, 4, 0, 0, 1,
          0, 0, 255, 4, 7, 5, 129, 3, 1, 0, 1, 9, 4, 1, 0, 2,
          0, 0, 255, 5, 7, 5, 2, 2, 16, 0, 0, 7, 5, 131, 2, 16,
          0, 0, 9, 4, 1, 1, 2, 0, 0, 255, 6, 7, 5, 2, 1, 16,
          0, 1, 7, 5, 131, 1, 16, 0, 1, 9, 4, 2, 0, 2, 0, 0,
          255, 5, 7, 5, 4, 2, 16, 0, 0, 7, 5, 133, 2, 16, 0, 0,
          9, 4, 2, 1, 2, 0, 0, 255, 6, 7, 5, 4, 1, 16, 0, 1,
          7, 5, 133, 1, 16, 0, 1, 9, 4, 3, 0, 2, 0, 0, 255, 5,
          7, 5, 6, 2, 16, 0, 0, 7, 5, 135, 2, 16, 0, 0, 9, 4,
          3, 1, 2, 0, 0, 255, 6, 7, 5, 6, 1, 16, 0, 1, 7, 5,
          135, 1, 16, 0, 1, 4, 3, 9, 4, 64, 3, 77, 0, 111, 0, 111,
          0, 114, 0, 101, 0, 32, 0, 67, 0, 111, 0, 109, 0, 112, 0, 117,
          0, 116, 0, 101, 0, 114, 0, 32, 0, 67, 0, 111, 0, 110, 0, 115,
          0, 117, 0, 108, 0, 116, 0, 97, 0, 110, 0, 116, 0, 115, 0, 44,
          0, 32, 0, 73, 0, 110, 0, 99, 0, 36, 3, 85, 0, 83, 0, 66,
```

```
        0, 32, 0, 68, 0, 101, 0, 109, 0, 111, 0, 32, 0, 70, 0, 105,
        0, 114, 0, 109, 0, 119, 0, 97, 0, 114, 0, 101, 0, 44, 3, 68,
        0, 101, 0, 102, 0, 97, 0, 117, 0, 108, 0, 116, 0, 32, 0, 99,
        0, 111, 0, 110, 0, 102, 0, 105, 0, 103, 0, 117, 0, 114, 0, 97,
        0, 116, 0, 105, 0, 111, 0, 110, 0, 38, 3, 73, 0, 110, 0, 116,
        0, 101, 0, 114, 0, 114, 0, 117, 0, 112, 0, 116, 0, 32, 0, 101,
        0, 110, 0, 100, 0, 112, 0, 111, 0, 105, 0, 110, 0, 116, 0, 20,
        3, 66, 0, 117, 0, 108, 0, 107, 0, 32, 0, 68, 0, 97, 0, 116,
        0, 97, 0, 18, 3, 73, 0, 83, 0, 79, 0, 32, 0, 68, 0, 97,
        0, 116, 0, 97, 0
        };
```

```
/* beginning of descriptor table */
RESPTRS gk_UsbResourceTable_1[] =
      {
      RESPTRVAL(gk_UsbResourceData, 0)
      };  /* end of gk_UsbResourceTable_1 */
```

Table of pointers to device descriptors.

```
/* beginning of descriptor table */
RESPTRS gk_UsbResourceTable_2[] =
      {
      RESPTRVAL(gk_UsbResourceData, 18)
      };  /* end of gk_UsbResourceTable_2 */
```

Table of pointers to configuration descriptors.

```
/* beginning of descriptor table */
RESPTRS gk_UsbResourceTable_3[] =
      {
      RESPTRVAL(gk_UsbResourceData, 181),
      RESPTRVAL(gk_UsbResourceData, 185),
      RESPTRVAL(gk_UsbResourceData, 249),
      RESPTRVAL(gk_UsbResourceData, 285),
      RESPTRVAL(gk_UsbResourceData, 329),
      RESPTRVAL(gk_UsbResourceData, 367),
      RESPTRVAL(gk_UsbResourceData, 387)
      };  /* end of gk_UsbResourceTable_3 */
```

Table of pointers to string descriptors for the default language.

Table of language mapping tables. This table has two entries, one for language ID 0x0409 (English, supplied by default); and another for language ID 0 (the default language). The two languages share the same table, which has 7 entries.

```
/* beginning of language ID data */
LANGIDPTRS gk_UsbLangidTable_4[] =
      {
      LANGPTRVAL(0x0409, 7, gk_UsbResourceTable_3),
      LANGPTRVAL(0x0000, 7, gk_UsbResourceTable_3)
      };  /* end of gk_UsbLangidTable_4 */

/**** the root table ****/
ROOTTABLE( \
      /* name */          gk_UsbDescriptorRoot, \
      /* devtbl, len devtbl */ gk_UsbResourceTable_1, 1, \
      /* cfgtbl, len cfgtbl */ gk_UsbResourceTable_2, 1, \
      /* strtbl, #langs */     gk_UsbLangidTable_4, 2, \
      /* others, len others */ NULL, 0 \
      );

/***** end of generated data *****/
```

5.3 A Multiple Interface Device, With Strings For Several Languages

```
# ex3.urc
#
# valid USB-resource-file with device descriptor,
# config descriptor and interface descriptors,
# set up to duplicate the slinky spec.  This is the
# same as ex2.urc, but with strings for several languages;
# it also illustrates the use of string tags for localization.
#
USB-resource-file 1.00 =
   {
   Device  {
       USB-version 1.00
       Class 0xFF  #vendor-specific
       SubClass 0xFF   #serial device
```

```
    Protocol 0x0 #none specified.
    Control-Packet-Size 8
    Vendor 0x40E %tag% MCCI                 MCCI Defined in Strings Section #1
    Product-ID 0xE101 %tag% PRODUCT         Product ID Defined in Strings
                                                Section #2
    Device-Version 1.00
    serial-number SERIALNUMBER              Serial Number Defined in Strings
                                                Section #3
    } %no label for device descriptor%;

#### here's the first configuration.
Configuration 1
    {
    %tag% CONFIG1_ID                            Config ID Defined in Strings
                                                Section #4
    Remote-Wakeup Self-Powered   %the flags%
    Power 0 mA
    % private descriptors would go here %

    % interface 0 has the interrupt endpoint %
    interface 0
        {
        % Alternate-Setting 0 %
               Class 0 #none defined.
                Subclass 0   %no subclass%
                 Protocol 0xFF   %vendor-specific%
                 name "Config 0/Ifc 0"   %stringtag%
                 % private descriptors %

            endpoints
                %only one:%
                interrupt in %address% 0x01 packet-size 1
                polling-interval 1
            ;
        }

    % interface 1 is used for sending data %
    interface 1
        {
        alternate-setting 0
            class 0     #none
            subclass 0x0 #none
            protocol 0xFF   #vendor-specific.
            name BULKDATA # string
            endpoints
                bulk out 2 packet-size 16
                bulk in 3 packet-size 16
            ;
        alternate-setting 1
```

```
        class 0     #none
        subclass 0  #none
        protocol 0xFF   #vendor-specific.
        name ISODATA# string
        endpoints
            isochronous out 2 packet-size 16
                polling-interval 1
            isochronous in 3 packet-size 16
                polling-interval 1
        ;
    }

% interface 2 is used for sending data %
interface 2
    {
    alternate-setting 0
        class 0     #none
        subclass 0x0#none
        protocol 0xFF   #vendor-specific.
        name BULKDATA # string
        endpoints
            bulk out 4 packet-size 16
            bulk in 5 packet-size 16
        ;
    alternate-setting 1
        class 0     #none
        subclass 0  #none
        protocol 0xFF   #vendor-specific.
        name ISODATA# string
        endpoints
            isochronous out 4 packet-size 16
                polling-interval 1
            isochronous in 5 packet-size 16
                polling-interval 1
        ;
    }

% interface 3 is used for sending data %
interface 3
    {
    alternate-setting 0
        class 0     #none
        subclass 0x0#none
        protocol 0xFF   #vendor-specific.
        name BULKDATA   # string
        endpoints
            bulk out 6 packet-size 16
            bulk in 7 packet-size 16
        ;
```

```
            alternate-setting 1
                class 0     #none
                subclass 0  #none
                protocol 0xFF   #vendor-specific.
                name ISODATA# string
                endpoints
                    isochronous out 6 packet-size 16
                        polling-interval 1
                    isochronous in 7 packet-size 16
                        polling-interval 1
                ;
            }

        } %no name% ;

    strings
        {
        default, language "English (USA)"
            {
            MCCI = "Moore Computer Consultants, Inc.";          String #1
            PRODUCT = "MCCI\xAE TrueTask\xAE USB Data Pump Demonstration
                Firmware";                                      String #2
            SERIALNUMBER = "00000000";                          String #3
            CONFIG1_ID = "MCCI USB Data Pump Loopback Configuration";
            "Config 0/Ifc 0" = "MCCI Loopback Control/Interrupt Interface";
            BULKDATA = "Bulk Data Loopback Interface";          String #4
            ISODATA = "Isochronous Data Loopback Interface";
            }
    # We provide string tables for translating traditional and simplified
    # Chinese We omit “language Chinese” because typically you have to choose
    # simplified or traditional Chinese depending on installed fonts.
        language "Chinese (traditional)"
            {

            MCCI =  ("\x{83AB}\x{6c0f}\x{96fb}\x{8166}"
                    "\x{9867}\x{554f}\x{516c}\x{53f8}");
```

莫氏電腦顧問公司

For convenience, we have written the string in two fragments, enclosed in parentheses. USBRC automatically concatenates the strings.

```
            PRODUCT = default;
            SERIALNUMBER = default;
            CONFIG1_ID = default;
            "Config 0/Ifc 0" = default;

            BULKDATA = "Bulk Data Loopback Interface";
            ISODATA = "Isochronous Data Loopback Interface";
            }

        # The same thing, for simplified Chinese.  Note that
        # the middle four characters are different, even though
```

```
            # they are pronounced the same.

            language "Chinese (simplified)"
                {
                MCCI =
                    "\x{83AB}\x{6c0f}\x{7535}\x{8111}"
                    "\x{987E}\x{95ee}\x{516c}\x{53f8}");
                PRODUCT = default;
                SERIALNUMBER = default;
                CONFIG1_ID = default;
                "Config 0/Ifc 0" = default;
                BULKDATA = "Bulk Data Loopback Interface";
                ISODATA = "Isochronous Data Loopback Interface";
                }
            } %end of strings; no label% ;
        } %end of resource data; no label% ;

    % end of file %
```

莫氏电脑顾问公司

Output file:

```
#include "usbrctab.h"

/* beginning of resource data */
RESCHAR gk_UsbResourceData[] =
      {
      18, 1, 0, 1, 255, 255, 0, 8, 14, 4, 1, 225, 0, 1, 1, 2,
      3, 1, 9, 2, 163, 0, 4, 1, 4, 96, 0, 9, 4, 0, 0, 1,
      0, 0, 255, 5, 7, 5, 129, 3, 1, 0, 1, 9, 4, 1, 0, 2,
      0, 0, 255, 6, 7, 5, 2, 2, 16, 0, 0, 7, 5, 131, 2, 16,
      0, 0, 9, 4, 1, 1, 2, 0, 0, 255, 7, 7, 5, 2, 1, 16,
      0, 1, 7, 5, 131, 1, 16, 0, 1, 9, 4, 2, 0, 2, 0, 0,
      255, 6, 7, 5, 4, 2, 16, 0, 0, 7, 5, 133, 2, 16, 0, 0,
      9, 4, 2, 1, 2, 0, 0, 255, 7, 7, 5, 4, 1, 16, 0, 1,
      7, 5, 133, 1, 16, 0, 1, 9, 4, 3, 0, 2, 0, 0, 255, 6,
      7, 5, 6, 2, 16, 0, 0, 7, 5, 135, 2, 16, 0, 0, 9, 4,
      3, 1, 2, 0, 0, 255, 7, 7, 5, 6, 1, 16, 0, 1, 7, 5,
      135, 1, 16, 0, 1, 8, 3, 9, 0, 4, 4, 4, 8, 66, 3, 77,
      0, 111, 0, 111, 0, 114, 0, 101, 0, 32, 0, 67, 0, 111, 0, 109,
      0, 112, 0, 117, 0, 116, 0, 101, 0, 114, 0, 32, 0, 67, 0, 111,
      0, 110, 0, 115, 0, 117, 0, 108, 0, 116, 0, 97, 0, 110, 0, 116,
      0, 115, 0, 44, 0, 32, 0, 73, 0, 110, 0, 99, 0, 46, 0, 106,
      3, 77, 0, 67, 0, 67, 0, 73, 0, 174, 0, 32, 0, 84, 0, 114,
      0, 117, 0, 101, 0, 84, 0, 97, 0, 115, 0, 107, 0, 174, 0, 32,
      0, 85, 0, 83, 0, 66, 0, 32, 0, 68, 0, 97, 0, 116, 0, 97,
      0, 32, 0, 80, 0, 117, 0, 109, 0, 112, 0, 32, 0, 68, 0, 101,
      0, 109, 0, 111, 0, 110, 0, 115, 0, 116, 0, 114, 0, 97, 0, 116,
      0, 105, 0, 111, 0, 110, 0, 32, 0, 70, 0, 105, 0, 114, 0, 109,
      0, 119, 0, 97, 0, 114, 0, 101, 0, 18, 3, 48, 0, 48, 0, 48,
      0, 48, 0, 48, 0, 48, 0, 48, 0, 48, 0, 84, 3, 77, 0, 67,
```

```
        0, 67, 0, 73, 0, 32, 0, 85, 0, 83, 0, 66, 0, 32, 0, 68,
        0, 97, 0, 116, 0, 97, 0, 32, 0, 80, 0, 117, 0, 109, 0, 112,
        0, 32, 0, 76, 0, 111, 0, 111, 0, 112, 0, 98, 0, 97, 0, 99,
        0, 107, 0, 32, 0, 67, 0, 111, 0, 110, 0, 102, 0, 105, 0, 103,
        0, 117, 0, 114, 0, 97, 0, 116, 0, 105, 0, 111, 0, 110, 0, 84,
        3, 77, 0, 67, 0, 67, 0, 73, 0, 32, 0, 76, 0, 111, 0, 111,
        0, 112, 0, 98, 0, 97, 0, 99, 0, 107, 0, 32, 0, 67, 0, 111,
        0, 110, 0, 116, 0, 114, 0, 111, 0, 108, 0, 47, 0, 73, 0, 110,
        0, 116, 0, 101, 0, 114, 0, 114, 0, 117, 0, 112, 0, 116, 0, 32,
        0, 73, 0, 110, 0, 116, 0, 101, 0, 114, 0, 102, 0, 97, 0, 99,
        0, 101, 0, 58, 3, 66, 0, 117, 0, 108, 0, 107, 0, 32, 0, 68,
        0, 97, 0, 116, 0, 97, 0, 32, 0, 76, 0, 111, 0, 111, 0, 112,
        0, 98, 0, 97, 0, 99, 0, 107, 0, 32, 0, 73, 0, 110, 0, 116,
        0, 101, 0, 114, 0, 102, 0, 97, 0, 99, 0, 101, 0, 72, 3, 73,
        0, 115, 0, 111, 0, 99, 0, 104, 0, 114, 0, 111, 0, 110, 0, 111,
        0, 117, 0, 115, 0, 32, 0, 68, 0, 97, 0, 116, 0, 97, 0, 32,
        0, 76, 0, 111, 0, 111, 0, 112, 0, 98, 0, 97, 0, 99, 0, 107,
        0, 32, 0, 73, 0, 110, 0, 116, 0, 101, 0, 114, 0, 102, 0, 97,
        0, 99, 0, 101, 0, 18, 3, 171, 131, 15, 108, 251, 150, 102, 129, 103,
        152, 79, 85, 108, 81, 248, 83, 18, 3, 171, 131, 15, 108, 53, 117, 17,
        129, 126, 152, 238, 149, 108, 81, 248, 83
        };

/* beginning of descriptor table */
RESPTRS gk_UsbResourceTable_1[] =
        {
        RESPTRVAL(gk_UsbResourceData, 0)
        };  /* end of gk_UsbResourceTable_1 */

/* beginning of descriptor table */
RESPTRS gk_UsbResourceTable_2[] =
        {
        RESPTRVAL(gk_UsbResourceData, 18)
        };  /* end of gk_UsbResourceTable_2 */

/* beginning of descriptor table */
RESPTRS gk_UsbResourceTable_3[] =
        {
        RESPTRVAL(gk_UsbResourceData, 181),
        RESPTRVAL(gk_UsbResourceData, 189),
        RESPTRVAL(gk_UsbResourceData, 255),
        RESPTRVAL(gk_UsbResourceData, 361),
        RESPTRVAL(gk_UsbResourceData, 379),
        RESPTRVAL(gk_UsbResourceData, 463),
        RESPTRVAL(gk_UsbResourceData, 547),
        RESPTRVAL(gk_UsbResourceData, 605)
        };  /* end of gk_UsbResourceTable_3 */

/* beginning of descriptor table */
```

This is the string table for English (also the default language for this device). Note that all the entries are the same as the other string tables, except for entry #1, which points to the English name for MCCI.

```
RESPTRS gk_UsbResourceTable_4[] =
      {
      RESPTRVAL(gk_UsbResourceData, 181),
      RESPTRVAL(gk_UsbResourceData, 677),
      RESPTRVAL(gk_UsbResourceData, 255),
      RESPTRVAL(gk_UsbResourceData, 361),
      RESPTRVAL(gk_UsbResourceData, 379),
      RESPTRVAL(gk_UsbResourceData, 463),
      RESPTRVAL(gk_UsbResourceData, 547),
      RESPTRVAL(gk_UsbResourceData, 605)
      };  /* end of gk_UsbResourceTable_4 */
```

This is the table for traditional Chinese. Note that all the entries are the same as the other string tables, except for entry #1, which points to the "traditional Chinese" UNICODE string for MCCI's Chinese name.

```
/* beginning of descriptor table */
RESPTRS gk_UsbResourceTable_5[] =
      {
      RESPTRVAL(gk_UsbResourceData, 181),
      RESPTRVAL(gk_UsbResourceData, 695),
      RESPTRVAL(gk_UsbResourceData, 255),
      RESPTRVAL(gk_UsbResourceData, 361),
      RESPTRVAL(gk_UsbResourceData, 379),
      RESPTRVAL(gk_UsbResourceData, 463),
      RESPTRVAL(gk_UsbResourceData, 547),
      RESPTRVAL(gk_UsbResourceData, 605)
      };  /* end of gk_UsbResourceTable_5 */
```

This is the table for simplified Chinese. Note that all the entries are the same as the other string tables, except for entry #1, which points to the "simplified Chinese" UNICODE string for MCCI's Chinese name.

```
/* beginning of language ID data */
LANGIDPTRS gk_UsbLangidTable_6[] =
      {
      LANGPTRVAL(0x0000, 8, gk_UsbResourceTable_3),
      LANGPTRVAL(0x0409, 8, gk_UsbResourceTable_3),
      LANGPTRVAL(0x0404, 8, gk_UsbResourceTable_4),
      LANGPTRVAL(0x0804, 8, gk_UsbResourceTable_5)
      };  /* end of gk_UsbLangidTable_6 */
```

The default language (0000) pointer
US English (0x409) pointer
Traditional Chinese (0x0404) pointer
Simplified Chinese (0x0804) pointer

```
/**** the root table ****/
ROOTTABLE( \
      /* name */          gk_UsbDescriptorRoot, \
      /* devtbl, len devtbl */ gk_UsbResourceTable_1, 1, \
      /* cfgtbl, len cfgtbl */ gk_UsbResourceTable_2, 1, \
      /* strtbl, #langs */     gk_UsbLangidTable_6, 4, \
      /* others, len others */ NULL, 0 \
      );
```

Pointer to language table; indicates that four distinct language IDs are supported (including language code 0, which is used for reading the list of known languages).

5.4 Multiple Configurations and Private Data

Finally, we present a grab-bag example that shows a number of different "extra features" of USBRC.

Input file:

```
# ex4.urc
#
# USB resource file with two configs, one little
# (from example 1) and one large (from example 2).
# Also has some private descriptors.
#

USB-resource-file 1.00 =
      {
      Device  {
          USB-version 1.00
          Class 0
          SubClass 0
          Protocol 0
          Control-Packet-Size 8
          Vendor 0x40E % could have a name here %
          Product-ID 0x1234 % could have a name here %
          Device-Version 2.05
          % could have a serial number here %
          } %no label% ;

      #### here's the first configuration.  Ideally, we
      #### ought to support autonum.
      Configuration 1
          {
          % string id would go here %
          Self-Powered Remote-Wakeup Bus-Powered   %the flags%
          Power 10 mA
          % private descriptors would go here %
          private-descriptors
              {
              raw {
                  %code% 0x55
                  %body -- just unicode% "I am a duck"
                  };
              }

          % interfaces go here %
          interface 0
              {
              % Alternate-Setting 0 %
                  Class 1
```

```
            Subclass 2
            Protocol 3
            % name {stringid} %
            % private descriptors %
            private-descriptors
                {
                raw {
                %code% 123
                %body -- just unicode% "Lots of luck"
                };
                }

            % endpoints would go here %
            endpoints
                interrupt in 1
                    packet-size 16
                    polling-interval 1
            ;
        }
    } %end config 1; no label% ;

#### here's the second configuration.
Configuration 2
    {
    % string id would go here %
    Remote-Wakeup Self-Powered   %the flags%
    Power 0 mA
    % private descriptors would go here %

    % interface 0 has the interrupt endpoint %
    interface 0
        {
        % Alternate-Setting 0 %
            Class 0     #none defined.
            Subclass 0  %no subclass%
            Protocol 0xFF   %vendor-specific%
            % name {stringid} %
            % private descriptors %

            endpoints
                %only one:%
                interrupt in %address% 0x01 packet-size 1
                polling-interval 1
            ;
        } %end interface%

    % interface 1 is used for sending data %
    interface 1
        {
```

```
    alternate-setting 0
        class 0     #none
        subclass 0x0#none
        protocol 0xFF   #vendor-specific.
        endpoints
            bulk out 2 packet-size 16
            bulk in 3 packet-size 16
        ;
    alternate-setting 1
        class 0     #none
        subclass 0  #none
        protocol 0xFF   #vendor-specific.
        endpoints
            isochronous out 2 packet-size 16
                polling-interval 1
            isochronous in 3 packet-size 16
                polling-interval 1
        ;
    } %end interface%

% interface 2 is used for sending data %
interface 2
    {
    alternate-setting 0
        class 0     #none
        subclass 0x0#none
        protocol 0xFF   #vendor-specific.
        endpoints
            bulk out 4 packet-size 16
            bulk in 5 packet-size 16
        ;
    alternate-setting 1
        class 0     #none
        subclass 0  #none
        protocol 0xFF   #vendor-specific.
        endpoints
            isochronous out 4 packet-size 16
                polling-interval 1
            isochronous in 5 packet-size 16
                polling-interval 1
        ;
    }

% interface 3 is used for sending data %
interface 3
    {
    alternate-setting 0
        class 0     #none
        subclass 0x0#none
```

```
                protocol 0xFF   #vendor-specific.
                endpoints
                    bulk out 6 packet-size 16
                    bulk in 7 packet-size 16
                ;
            alternate-setting 1
                class 0     #none
                subclass 0  #none
                protocol 0xFF   #vendor-specific.
                endpoints
                    isochronous out 6 packet-size 16
                        polling-interval 1
                    isochronous in 7 packet-size 16
                        polling-interval 1
                ;
            } %end interface%
        } %end config 2, no label% ;
    } %end device, no label% ;
% end of file %
```

Output file:

```
/* <<commentary omitted for brevity>> */

#include "usbrctab.h"

/* beginning of resource data */
RESCHAR gk_UsbResourceData[] =
    {
    18, 1, 0, 1, 0, 0, 0, 8, 14, 4, 52, 18, 5, 2, 0, 0,
    0, 2, 9, 2, 75, 0, 1, 1, 0, 224, 5, 24, 85, 73, 0, 32,
    0, 97, 0, 109, 0, 32, 0, 97, 0, 32, 0, 100, 0, 117, 0, 99,
    0, 107, 0, 9, 4, 0, 0, 1, 1, 2, 3, 0, 26, 123, 76, 0,
    111, 0, 116, 0, 115, 0, 32, 0, 111, 0, 102, 0, 32, 0, 108, 0,
    117, 0, 99, 0, 107, 0, 7, 5, 129, 3, 16, 0, 1, 9, 2, 163,
    0, 4, 2, 0, 96, 0, 9, 4, 0, 0, 1, 0, 0, 255, 0, 7,
    5, 129, 3, 1, 0, 1, 9, 4, 1, 0, 2, 0, 0, 255, 0, 7,
    5, 2, 2, 16, 0, 0, 7, 5, 131, 2, 16, 0, 0, 9, 4, 1,
    1, 2, 0, 0, 255, 0, 7, 5, 2, 1, 16, 0, 1, 7, 5, 131,
    1, 16, 0, 1, 9, 4, 2, 0, 2, 0, 0, 255, 0, 7, 5, 4,
    2, 16, 0, 0, 7, 5, 133, 2, 16, 0, 0, 9, 4, 2, 1, 2,
    0, 0, 255, 0, 7, 5, 4, 1, 16, 0, 1, 7, 5, 133, 1, 16,
    0, 1, 9, 4, 3, 0, 2, 0, 0, 255, 0, 7, 5, 6, 2, 16,
    0, 0, 7, 5, 135, 2, 16, 0, 0, 9, 4, 3, 1, 2, 0, 0,
    255, 0, 7, 5, 6, 1, 16, 0, 1, 7, 5, 135, 1, 16, 0, 1
    };

/* beginning of descriptor table */
RESPTRS gk_UsbResourceTable_1[] =
```

```
        {
        RESPTRVAL(gk_UsbResourceData, 0)
        };  /* end of gk_UsbResourceTable_1 */

/* beginning of descriptor table */
RESPTRS gk_UsbResourceTable_2[] =
        {
        RESPTRVAL(gk_UsbResourceData, 18),
        RESPTRVAL(gk_UsbResourceData, 93)
        };  /* end of gk_UsbResourceTable_2 */
```

Table of pointers to configuration descriptors.

```
/**** the root table ****/
ROOTTABLE( \
        /* name */          gk_UsbDescriptorRoot, \
        /* devtbl, len devtbl */ gk_UsbResourceTable_1, 1, \
        /* cfgtbl, len cfgtbl */ gk_UsbResourceTable_2, 2, \
        /* strtbl, #langs */     NULL, 0, \
        /* others, len others */ NULL, 0 \
        );

/***** end of generated data *****/
```

5.5 Sample DataPump section

This section gives an example of the DataPump section that is appropriate for the Agere USS820. This information is normally passed in using the "`-chipinfo`" switch; the information here must match the conventions used by the chip driver. The chipinfo file for a given chip is normally found in `ifc/`*{chip}*`/common/chipinfo.urc`

Input file:

```
# ex5.urc
#
# Data Pump Information for USBRC #
#

DataPump
        {
        Chip-Name           "Uss820"
        Include-File-Name       "u820kern.h"
        Chip-Header-Name    "UDEV820_HDR"
        Chip-Data-Structure-Name "UDEV820"
        Init-Function-Name      "UDEV820Init"
        Endpoint-Data-Structure-Name "EPIO820"
        # Config-Data-Structure-Name "UCFG820"
        # Interface-Data-Structure-Name  "UIFCSET820"
        # Setting-Data-Structure-Name    "UIFC820"
        Number-of-Endpoints     16
```

```
    % The only requirement for the mapping is that, %
    % for each endpoint address, there is only one  %
    % corresponding endpoint index            %
    Endpoint-Mapping
        {
        ( 0,    0)
        ( 0x80, 1)
        ( 1,2)
        ( 0x81, 3)
        ( 2,4)
        ( 0x82, 5)
        ( 3,6)
        ( 0x83, 7)
        ( 4,8)
        ( 0x84, 9)
        ( 5,10)
        ( 0x85, 11)
        ( 6,12)
        ( 0x86, 13)
        ( 7,    14)
        ( 0x87, 15)
        }
};
```

5.6 Sample Application Section

This section presents an example of a correctly completed **application** section.

```
application
   {
   descriptor table
        {
        root name "test37_root";
        internal names static;
        functions
            {
            with string filter table;
            get-descriptor "testapp_get_descriptor";
            set-descriptor "testapp_set_descriptor";
            };
        };
   header-file
        {
        file name "test37dh.h";
        with string ids;
        };
   initialization
        {
        % assumed name of file %
```

```
            file name "initfile.c";

            % name to compile for the init function: %
            function name "test37_init_function";
            };
        strings # info for writing the string header file.
            {
            % assumed name of output file %
            file name "test37st.h";

            % prefix for emitted string ids %
            prefix "TEST37STRING";
            };
        structure
            {
            type name "TEST37_TYPE";
            };

        };
```

5.7 Sample CDC USB Ethernet Adapter

The following example shows how to use USBRC to generate the descriptors for a USB Ethernet adapter. This example is based on the CDC 1.1 spec.

```
USB-resource-file 1.00 =
   {
   Device  {
       USB-version 1.1
       Class 0xFF  #vendor-specific
       SubClass 0xFF   #serial device
       Protocol 0x0 #none specified.
       Control-Packet-Size 64
       Vendor 0x40E %tag% VENDOR_ID  # MCCI's vendor ID.
       Product-ID 0xF102 %tag% PRODUCT
       Device-Version 1.10
       serial-number SERIALNUMBER
       } %no external name% ;

   #### here's the first configuration.
   Configuration 1
       {
       %tag% CONFIG1_ID
       Bus-Powered Self-Powered %the flags%
       Power 2 mA
       % config private descriptors would go here %
```

```
#
# INTERFACE 0 is the COMM CLASS interface.  We have assigned
#       a vendor-specific sub-class.  However, we can still use
#       the UNION descriptor to bind the DATA interface to this
#       COMM interface.
#
interface 0
    {
    class 2         #comm class
    subclass 0x06   #ethernet control model
    protocol 0x0    #no protocol
    name COMMIFC    # string

    # class descriptors follow the interface header.  Since USBRC
    # doesn't know how to compile CDC descriptors, we have to code
    # them by hand.
    private-descriptors
        {
        # CDC header
        raw {
            0x24# interface functional descriptor
            0   # functional descriptor: header
            # the version of the CDC specification:
            word(0x110)
            };

        # CDC union descriptor
        raw {
            0x24# interface
            6       # 'union'
            % interface number of control interface % 0
            % interface number of data interface % 1
            };

        # CDC ethernet control model functional descriptor
        raw {
            0x24    # interface
            0x0F    # ethernet
            % mac address string index %
            string-index MACADDR
            dword(0)# supported statistics bitmap
            word(1514)  # max seg size
            word(0x8040)# 64 imperfect
                        #   multicast filters
            0       # bNumberPowerFilters
            };
        }

    #### the communication-class interface has a notification endpoint
```

```
            endpoints
                interrupt in 1 packet-size 64
                    polling-interval 10 %ms%
                ;
            }

        #
        # INTERFACE 1 is the DATA CLASS interface.  In addition to setting 0,
        #       which must have no endpoints, there are two alternate
        #       settings, one using BULK to transfer data, and another using
        #       BULK to transfer data using vendor-specific wrappers.
        #
        interface 1
            {
            alternate-setting 0
                class 0xA   #data class
                subclass 0  #normal data class
                protocol 0  #vendor-specific.
                name RESET  # string
                % no endpoints %
                ;
            alternate-setting 1
                class 0xA   #data class
                subclass 0  #normal data class
                protocol 0  #vendor-specific.
                name BULKDATA   # string
                endpoints
                    bulk out 2 packet-size 64
                    bulk in 2 packet-size 64
                ;
            alternate-setting 2
                class 0xA   #data class
                subclass 0  #normal data class
                protocol 0xFF   #vendor-specific.
                name WRAPPERDATA# string
                endpoints
                    bulk out 2 packet-size 64
                    bulk in 2 packet-size 64
                ;
            }
        };

    #
    #  In this example, we want to get the serial number and the MAC address
    #  from an external EEPROM.  (We could have used exactly the same index,
    #  but we're assuming that the SERIALNUMBER might be shorter than the
    #  MAC address, to accommodate operating system limitations.)  Anyway,
    #  we've coded them separately in this example.
    #
```

```
        strings
            {
            properties external SERIALNUMBER, MACADDR;

            default, language "English (USA)"
                {
                VENDOR_ID = "Moore Computer Consultants, Inc.";
                PRODUCT = "Skimmer Ethernet Adapter";
                SERIALNUMBER = "19980906a";
                MACADDR = "(dummy)";     # none by default
                CONFIG1_ID = "MCCI(r) CDC Ethernet Prototype Configuration";
                RESETDATA = "Data Interface, Reset";
                BULKDATA = "Data Interface, Bulk Mode";
                MIXEDDATA = "Data Interface, Wrapper (MCCI) Mode";
                }
            } %strings% ;
        } %no name% ;
```

5.8 A Simple Device with OTG

Input file:

```
    # ex8.urc
    #
    # USB-resource-file with device descriptor, OTG,
    # config descriptor and interface descriptor, with
    # first setting (but nothing else)
    #
    # Test-Options:

    USB-resource-file 1.00 =
          {
          Device  {

              USB-version 2.00
              Class 0
              SubClass 0
              Protocol 0
              Control-Packet-Size 8
              Vendor 0x040E % string name omitted %
              Product-ID 0x1234 % name omitted %
              Device-Version 2.05
              % serial-number omitted %
              } %no label for device% ;

          on-the-go hnp ;

    #### here's the first (and only) configuration.
          Configuration 1
```

```
        {
            % configuration name would go here %
            Self-Powered Remote-Wakeup Bus-Powered
                %the flags%
            Power 10 mA
        % private descriptors would go here %
        # Next we have the (only) interface
            interface 0
                {
        % Alternate-Setting 0 %
                    Class 1
                    Subclass 2
                    Protocol 3
                    % name {stringid} %
                    % private descriptors %
                    % endpoints for this setting %

                    endpoints
                        %only one:%
                        interrupt
                        in
                        %address% 0x01
                        packet-size 1
                        polling-interval 1
                    %more endpoints would be here%
                    ;
                }
            } %no label on the configuration descriptor% ;
        } %no label on the overall resource file% ;
% end of file %
```

Output file (generated with `-dpapi 0`):

```
/* <<commentary omitted for brevity>> */
#include "usbrctab.h"

/* beginning of resource data */
RESCHAR gk_UsbResourceData[] =
        {
    18, 1, 0, 1, 0, 0, 0, 8,
    14, 4, 52, 18, 5, 2, 0, 0,
    0, 1, 9, 2, 25, 0, 1, 1,
    0, 224, 5, 3, 9, 3, 9, 4,
    0, 0, 1, 1, 2, 3, 0, 7,
    5, 129, 3, 1, 0, 1
        };
```

Device descriptor data

Configuration 1 descriptor data

Interface 0 descriptor data

OTG descriptor data

Endpoint descriptor data

```
/* beginning of descriptor table */
RESPTRS gk_UsbResourceTable_1[] =
```

Table of pointers to device descriptors.

```
      {
      RESPTRVAL(gk_UsbResourceData, 0)
      };  /* end of gk_UsbResourceTable_1 */
```

Table of pointers to configuration descriptors.

```
/* beginning of descriptor table */
RESPTRS gk_UsbResourceTable_2[] =
      {
      RESPTRVAL(gk_UsbResourceData, 18)
      };  /* end of gk_UsbResourceTable_2 */
```

Table of pointers to tables of descriptors. Generated via a macro so you can control its layout.

```
/**** the root table ****/
ROOTTABLE( \
      /* name */           gk_UsbDescriptorRoot, \
      /* devtbl, len devtbl */ gk_UsbResourceTable_1, 1, \
      /* cfgtbl, len cfgtbl */ gk_UsbResourceTable_2, 1, \
      /* strtbl, #langs */     NULL, 0, \
      /* others, len others */ NULL, 0 \
      );

/***** end of generated data *****/
```

5.9 A Simple Device with High speed / Expressions

Input file:

```
# ex9.urc
#
# USB-resource-file with device descriptor and
# config descriptor and interface descriptor, with
# first setting (but nothing else)
#

USB-resource-file 1.00 =
      {
      Device  {
```

```
            USB-version 1.00
            Class 0
            SubClass 0
            Protocol 0
            Control-Packet-Size 8
            Vendor 0x040E % string name omitted %
            Product-ID 0x1234 % name omitted %
            Device-Version 2.05
            % serial-number omitted %
            } %no label% ;

        #### here's the first (and only) configuration.
        Configuration 1
            {
            % configuration name would go here %
            Self-Powered Remote-Wakeup Bus-Powered   %the flags%
            Power 10 mA
            % private descriptors would go here %

            # next we have the (only) interface:
            interface 0
                {
                % Alternate-Setting 0 %
                    Class 1
                    Subclass 2
                    Protocol 3
                    % name {omitted} %
                    % private descriptors %

                    % endpoints for this setting %
                    endpoints
                        bulk out 1 packet-size (speed == full ? 16 : 512)
                        bulk in  1 packet-size (speed == full ? 16 : 512)

                        %more endpoints would go here%
                    ;
                }
            } %no label for config statement% ;
        } %no label for resource file% ;

    % end of file %

    % DataPump Information for USBRC %
    Data-Pump
        {
        Chip-Name           "ivtra"
        Include-File-Name       "ivtrakern.h"
        Chip-Header-Name    "UDEVIVTRA_HDR"
        Chip-Data-Structure-Name "UDEVIVTRA"
```

```
        Init-Function-Name      "UDEVIVTRAInit"
        Endpoint-Data-Structure-Name "UEPIVTRA"

        Number-of-Endpoints     2

        speeds full high

        % The only requirement for the mapping is that, %
        % for each endpoint address, there is only one  %
        % corresponding endpoint index                  %
        Endpoint-Mapping
            {
            ( 0,       0)
            ( 0x80,    1)
            }
        };

    % end of file %
```

Output file (generated with `-dpapi 2`):

```
    /*

    Module: ex6_new.c

    Function:
       USB peripheral device descriptor data, generated automatically
       from ex6_new.urc by v223m, version V2.23m (Jan 19 2006)

    Copyright:
       Copyright 1997-2006, Moore Computer Consultants, Inc.

       Distribution and use in source and binary forms, with or without
       modification, are permitted provided that the following conditions
       are met:

       1. Redistribution of source code must retain the above copyright
          notice, this list of conditions and the following disclaimer.
       2. MCCI's name may not be used to endorse or promote products
          derived from this software without specific prior written
          permission.

    Disclaimer:
       THIS FILE IS PROVIDED BY MCCI "AS IS" AND ANY EXPRESS OR
       IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
       WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
       ARE DISCLAIMED.  IN NO EVENT SHALL MCCI BE LIABLE FOR ANY
       DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
       DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE
       GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
```

```
   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
   WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
   NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
   THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Generated:
   Thursday, January 19, 2006   5:03 pm

*/

/* make sure we get the V3 structures */
#ifndef __USBRCTAB_WANT_V3
# define __USBRCTAB_WANT_V3 1
#endif /* __USBRCTAB_WANT_V3 */

#include "{no-header-file-name-given}"

/* beginning of resource data */
RESCHAR gk_UsbResourceData[] =
   {
   18, 1, 0, 1, 0, 0, 0, 8,
   14, 4, 52, 18, 5, 2, 0, 0,
   0, 1, 9, 2, 32, 0, 1, 1,
   0, 224, 5, 9, 4, 0, 0, 2,
   1, 2, 3, 0, 7, 5, 1, 2,
   16, 0, 0, 7, 5, 129, 2, 16,
   0, 0, 18, 1, 0, 1, 0, 0,
   0, 8, 14, 4, 52, 18, 5, 2,
   0, 0, 0, 1, 9, 2, 32, 0,
   1, 1, 0, 224, 5, 9, 4, 0,
   0, 2, 1, 2, 3, 0, 7, 5,
   1, 2, 0, 2, 0, 7, 5, 129,
   2, 0, 2, 0
   };

/* beginning of descriptor table */
RESPTRS gk_UsbResourceTable_1[] =
   {
   RESPTRVAL(gk_UsbResourceData, 0)
   };  /* end of gk_UsbResourceTable_1 */

/* beginning of descriptor table */
RESPTRS gk_UsbResourceTable_2[] =
   {
   RESPTRVAL(gk_UsbResourceData, 18)
   };  /* end of gk_UsbResourceTable_2 */

/* beginning of descriptor table */
RESPTRS gk_UsbResourceTable_3[] =
```

```
   {
   RESPTRVAL(gk_UsbResourceData, 50)
   };  /* end of gk_UsbResourceTable_3 */

/* beginning of descriptor table */
RESPTRS gk_UsbResourceTable_4[] =
   {
   RESPTRVAL(gk_UsbResourceData, 68)
   };  /* end of gk_UsbResourceTable_4 */


/**** init function declaration ****/
USBRCTAB_DECLARE_DCD_INIT_FN_NOCLASS(UDEVIVTRAInit)

/**** the root table ****/
ROOTTABLE_V3_WITH_SEMI( \
   /* name */          gk_UsbDescriptorRoot, \
   /* init-device-fn */     UDEVIVTRAInit, \
   /* UDEVICE size */       sizeof(UDEVIVTRA), \
   /* devtbl, len devtbl */ gk_UsbResourceTable_1, 1, \
   /* cfgtbl, len cfgtbl */ gk_UsbResourceTable_2, 1, \
   /* hsdevtbl, len hsdevtbl */ gk_UsbResourceTable_3, 1, \
   /* hscfgtbl, len hscfgtbl */ gk_UsbResourceTable_4, 1, \
   /* pOsFeatureTable */        0, \
   /* strtbl, #langs */     0, 0, \
   /* others, len others */ 0, 0, \
   /* get-descriptor-fn */      0, \
   /* set-descriptor-fn */      0, \
   /* ext bitmap, len */        0, 0, \
   /* r/o bitmap, len */        0, 0 \
   )


/***** end of generated data *****/
```